Zawody II stopnia

Zawody II stopnia — opracowania zadań

Wojciech Guzicki, Jarosław Wróblewski — Treść zadania
Wojciech Guzicki — Opracowanie
Marcin Mucha — Program

Gorszy Goldbach

W roku 1742 C. Goldbach w liście do L. Eulera napisał, że jego zdaniem każda liczba całkowita $n > 5$ jest sumą trzech liczb pierwszych[1].

Euler odpisał, że jest to równoważne temu, że każda liczba parzysta $n \geq 4$ jest sumą dwóch liczb pierwszych. To jednak nie przybliżyło ich do rozwiązania podstawowego problemu: czy tak jest naprawdę.

Dziś wiemy, że jest tak dla liczb aż do $4 \cdot 10^{11}$ (wiemy też dużo więcej, ale cała hipoteza jest nadal problemem otwartym). Nie będziemy tego sprawdzać, postawimy sobie mniej ambitne zadanie. Okazuje się, że każda liczba naturalna $n \geq 10$ jest sumą różnych nieparzystych liczb pierwszych.

Zadanie

Twoje zadanie polega na napisaniu programu, który:

- *wczyta z pliku tekstowego* `gol.in` *liczby całkowite,*
- *znajdzie ich rozkłady na sumy różnych nieparzystych liczb pierwszych,*
- *zapisze wyniki w pliku tekstowym* `gol.out`.

Takich rozkładów może być wiele. Twój program może znaleźć jakikolwiek z nich.

Wejście

W pierwszym wierszu pliku tekstowego `gol.in` *zapisano jedną dodatnią liczbę całkowitą n, $n \leq 40$. W każdym z kolejnych n wierszy znajduje się jedna liczba całkowita z przedziału $[10, \ldots, 2\,000\,000\,000]$.*

Wyjście

Rozkład liczby k musi być zapisany w dwóch wierszach. W pierwszym wierszu należy zapisać jedną liczbę całkowitą $m \geq 1$, będącą liczbą składników rozkładu.

W drugim wierszu należy zapisać, w rosnącej kolejności, m różnych nieparzystych liczb pierwszych, których suma jest równa k, pooddzielanych pojedynczymi odstępami. Rozkłady powinny występować w kolejności zgodnej z kolejnością liczb w pliku wejściowym.

Przykład

Dla pliku wejściowego `gol.in`*:*

```
2
59
15
```

poprawną odpowiedzią jest plik wyjściowy `gol.out`*:*

```
5
5 7 11 17 19
3
3 5 7
```

Rozwiązanie

Istnieje bardzo szybki algorytm pozwalający rozwiązać to zadanie. Wskażemy mianowicie 32 nieparzyste liczby pierwsze o tej własności, że każda dopuszczalna liczba n (tzn. z przedziału $[10; 2\,000\,000\,000]$) będzie sumą niektórych z tych liczb. Pokażemy też dokładnie, jak taki rozkład można znaleźć.

[1] Liczba pierwsza to liczba naturalna $n > 1$, która ma tylko dwa dzielniki naturalne: 1 oraz n.

Algorytm, który mamy na myśli, można odczytać z dowodu następującego twierdzenia ogólnego, udowodnionego w 1949 roku przez H. E. Richerta.

Twierdzenie. Każda liczba naturalna $n \geq 10$ jest sumą różnych nieparzystych liczb pierwszych.

Dowód. Najpierw przedstawimy każdą liczbą naturalną n spełniającą nierówności

$$10 \leq n \leq 46$$

w postaci sumy różnych nieparzystych liczb pierwszych wybranych spośród liczb 3, 5, 7, 11, 13, 17. A oto te rozkłady:

10	=	3+7,			
11	=	11,	29	=	5+7+17,
12	=	5+7,	30	=	13+17,
13	=	13,	31	=	3+11+17,
14	=	3+11,	32	=	3+5+7+17,
15	=	3+5+7,	33	=	5+11+17,
16	=	5+11,	34	=	3+7+11+13,
17	=	17,	35	=	5+13+17,
18	=	5+13,	36	=	3+5+11+17,
19	=	3+5+11,	37	=	7+13+17,
20	=	3+17,	38	=	3+5+13+17,
21	=	3+5+13,	39	=	3+5+7+11+13,
22	=	5+17,	40	=	5+7+11+17,
23	=	5+7+11,	41	=	11+13+17,
24	=	7+17,	42	=	5+7+13+17,
25	=	3+5+17,	43	=	3+5+7+11+17,
26	=	3+5+7+11,	44	=	3+11+13+17,
27	=	3+7+17,	45	=	3+5+7+13+17,
28	=	11+17,	46	=	5+11+13+17

Tabela 1.

Tych liczb jest 37. Wybieramy teraz największą liczbę pierwszą nie większą od 37. Będzie to sama liczba 37. Teraz do każdego z otrzymanych 37 rozkładów dołączamy liczbę 37. W ten sposób otrzymamy rozkłady liczb od $10 + 37 = 47$ do $46 + 37 = 83$ na sumy różnych nieparzystych liczb pierwszych. Łącznie z otrzymanymi wcześniej rozkładami mamy teraz rozkłady 74 liczb: od 10 do 83.

Wybieramy następną liczbę pierwszą: będzie to największa liczba pierwsza nie większa od 74; w naszym przypadku jest to liczba 73. Dołączamy tę liczbę do każdego z dotychczasowych rozkładów, otrzymując w ten sposób rozkłady wszystkich liczb aż do $83 + 73 = 156$. Mamy więc rozkłady 147 liczb od 10 do 156.

Wybieramy tym razem największą liczbę pierwszą nie większą od 147, dołączamy ją do znalezionych rozkładów i tak dalej.

Jest to bardzo naturalna metoda postępowania. Powstaje jednak pytanie, czy jest ona zawsze skuteczna. Mianowicie mogłoby się okazać, że kolejna wybrana liczba pierwsza (pamiętamy: wybieramy największą liczbę nie większą od liczby tych liczb, dla których już znaleźliśmy rozkład) jest równa poprzednio wybranej liczbie pierwszej. Inaczej mówiąc, mogłoby się okazać, że za ostatnio wybraną liczbą pierwszą występuje tak wiele liczb złożonych, że nie możemy dobrać odpowiednio kolejnej liczby pierwszej. Okazuje się jednak, że tak nie jest i można tego dowieść. Dowód wymaga jednak skorzystania z bardzo znanego twierdzenia z teorii liczb, tzw. twierdzenia Czebyszewa, które zapewne nie jest znane większości uczniów liceum. W naszym zadaniu jednak dowód twierdzenia nie jest potrzebny. Wystarczy bowiem sprawdzić, że taką liczbę można dobrać zawsze do momentu, gdy uzyskamy rozkłady wszystkich liczb nie większych od 2 miliardów. Przykłady takich liczb pierwszych zawarte są w tabeli 2.

W kolumnie pierwszej podajemy największą dotychchczas użytą liczbę pierwszą p. W kolumnie drugiej podajemy przedział liczb, w którym ta liczba p jest użyta jako największa liczba rozkładu (z wyjątkiem pierwszego wiersza). Wreszcie w kolumnie trzeciej podajemy, ile liczb ma już znany rozkład (od 10 do największej liczby, dla której mamy znaleziony rozkład). Ta liczba jest o 9 mniejsza od największej liczby, dla której znamy rozkład.

Zwróćmy jeszcze uwagę na to, że w pierwszym wierszu występuje liczba pierwsza 17 jako największa liczba występująca w rozkładach liczb od 10 do 46. To nie znaczy oczywiście, że występuje ona w każdym z tych rozkładów. Rozkłady liczb od 10 do 46 muszą być brane z tabeli pierwszej.

Ostatnia liczba pierwsza p	przedział	Ile liczb (razem)
17	$\langle 10; 46\rangle$	37
37	$\langle 47; 83\rangle$	74
73	$\langle 75; 156\rangle$	147
139	$\langle 157; 295\rangle$	286
283	$\langle 296; 578\rangle$	569
569	$\langle 579; 1147\rangle$	1138
1129	$\langle 1148; 2276\rangle$	2267
2267	$\langle 2277; 4543\rangle$	4534
4523	$\langle 4544; 9066\rangle$	9057
9049	$\langle 9067; 18115\rangle$	18106
18097	$\langle 18116; 36212\rangle$	36203
36191	$\langle 36213; 72403\rangle$	72394
72383	$\langle 72404; 144786\rangle$	144777
144773	$\langle 144787; 289559\rangle$	289550
289543	$\langle 289560; 579102\rangle$	579093
579083	$\langle 579103; 1158185\rangle$	1158176
1158161	$\langle 1158186; 2316346\rangle$	2316337
2316337	$\langle 2316347; 4632683\rangle$	4632674
4632673	$\langle 4632684; 9265356\rangle$	9265347
9265309	$\langle 9265357; 18530665\rangle$	18530656
18530639	$\langle 18530666; 37061304\rangle$	37061295
37061293	$\langle 37061305; 74122597\rangle$	74122588
74122571	$\langle 74122598; 148245168\rangle$	148245159
148245127	$\langle 148245169; 296490295\rangle$	296490286
296490277	$\langle 296490296; 592980572\rangle$	592980563
592980559	$\langle 592980573; 1185961131\rangle$	1185961122
1185961087	$\langle 1185961132; 2000000000\rangle$	1999999991

$Tabela 2.$

Teraz już można łatwo napisać algorytm:

Dana jest liczba $n \geq 10$.
Dopóki $n > 46$ **powtarzaj**
  Znajdź w drugiej kolumnie przedział, do którego należy n.
  Zapamiętaj odpowiadającą mu liczbę p *z pierwszej kolumny.*
  $n := n - p$.
Teraz już $n \leq 46$.
Zapamiętaj znajdujące się w tabeli pierwszej liczby pierwsze występujące w rozkładzie n.
Zapamiętane liczby wypisz w kolejności rosnącej.

Powróćmy teraz do dowodu twierdzenia. Oznaczmy przez p największą użytą liczbę pierwszą i przez M największą liczbę, dla której znaleźliśmy rozkład. Niech k będzie liczbą tych liczb, dla których taki rozkład znaleźliśmy. Oczywiście wtedy $k = M - 9$.

Na początku mamy $p = 17$, $M = 46$ i $k = 37$. Zauważmy, że

$$k \geq 2p.$$

Niech teraz q będzie największą liczbą pierwszą nie większą od k. Wtedy $q > p$. Wynika to ze znanego twierdzenia Czebyszewa:

Twierdzenie. Dla każdej liczby naturalnej $l \geq 2$ istnieje liczba pierwsza r taka, że $l < r < 2l$.

Dowód twierdzenia Czebyszewa można znaleźć np. w książkach W. Sierpińskiego *Arytmetyka teoretyczna* lub *Teoria liczb, t. II.* Z twierdzenia Czebyszewa wynika, że istnieje nieparzysta liczba pierwsza r taka, że:

$$p < r < 2p \leq k.$$

Liczba q, która jest największą liczbą pierwszą nie większą od k, jest więc większa od p. Teraz do q ostatnich otrzymanych liczb dodajemy q:

$$\begin{aligned}
(M-q+1)+q &= M+1,\\
(M-q+2)+q &= M+2,\\
&\dots\ \dots\ \dots\\
(M-2)+q &= M+q-2,\\
(M-1)+q &= M+q-1,\\
M+q &= M+q.
\end{aligned}$$

Otrzymaliśmy rozkłady liczb od $M+1$ do $M+q$ na sumę różnych liczb pierwszych: liczba q jest bowiem większa od wszystkich dotychczas użytych liczb pierwszych. Oczywiście również $M+q > M$.

Połóżmy teraz

$$M_1 = M+q, \quad p_1 = q, \quad k_1 = M_1 - 9.$$

Ponieważ $q \leq k$, więc

$$2q \leq k+q = M-9+q = M_1 - 9 = k_1.$$

Podstawiamy

$$M := M_1, \quad p := p_1, \quad k := k_1$$

i mamy nadal spełniony warunek $2p \leq k$. Możemy więc powtórzyć postępowanie itd.

Ponieważ $M_1 > M$, więc za każdym razem dostajemy rozkład co najmniej jednej nowej liczby na sumę różnych nieparzystych liczb pierwszych, co dowodzi, że każda liczba $n \geq 10$ ma taki rozkład. To kończy dowód twierdzenia Richerta.

Można podać wiele innych algorytmów rozwiązujących to zadanie. Najprostszy polega na wybieraniu największej liczby pierwszej p nie większej od n i następnie rozwiązaniu zadania dla liczby $n-p$ zamiast n. Ten algorytm nie zawsze prowadzi do sukcesu. Jeśli np. $n = 27$, to wybierzemy liczbę $p = 23$ i wtedy zadanie nie ma rozwiązania dla $n - p = 4$. Możemy jednak próbować zadbać o to, by otrzymana różnica była co najmniej równa 10: wtedy zadanie dla $n-p$ będzie miało rozwiązanie. To daje bardzo prosty algorytm:

Dana jest liczba $n \geq 10$.
Dopóki $n \geq 10$ **powtarzaj**
  Znajdź liczbę pierwszą p taką, że $\frac{n}{2} < p \leq n-10$.
  Zapamiętaj liczbę p.
  $n := n - p$.
Zapamiętane liczby pierwsze wypisz w kolejności rosnącej.

Warunek $\frac{n}{2} < p$ gwarantuje, że kolejno znajdowane liczby pierwsze będą różne. Mamy bowiem wtedy

$$n - p < \frac{n}{2} < p,$$

a więc następna liczba pierwsza będzie mniejsza od p.

Ten algorytm jednak nie jest poprawny, gdyż nie dla każdej liczby n istnieje liczba pierwsza p taka, że

$$\frac{n}{2} < p \leq n - 10.$$

Jednak taka liczba pierwsza p istnieje dla odpowiednio dużych liczb n. Niewielka modyfikacja dowodu twierdzenia Czebyszewa daje następujące wzmocnienie tego twierdzenia:

Twierdzenie. Jeśli $n \geq 21$, to w przedziale $\langle n+1; 2n \rangle$ istnieje co najmniej 5 liczb pierwszych.

Z tego twierdzenia wynika łatwo następujący wniosek:

Wniosek. Jeśli $n > 26$, to istnieje liczba pierwsza p taka, że

$$\frac{n}{2} < p \leq n - 10.$$

Zauważmy, że z tego wniosku wynika natychmiast inny dowód twierdzenia Richerta. Przeprowadzenie tego dowodu pozostawimy jako ćwiczenie dla Czytelnika. Możemy też zmodyfikować algorytm, tak by działał poprawnie dla wszystkich n:

Dana jest liczba $n \geq 10$.
Dopóki $n > 26$ **powtarzaj**
  Znajdź liczbę pierwszą p *taką, że* $\frac{n}{2} < p \leq n - 10$.
  Zapamiętaj liczbę p.
  $n := n - p$
Teraz już $n \leq 26$.
Zapamiętaj znajdujące się w tabeli pierwszej liczby pierwsze występujące w rozkładzie n.
Zapamiętane liczby pierwsze wypisz w kolejności rosnącej.

Ten algorytm może jednak działać dość długo. Liczby p najlepiej szukać „od góry": po odjęciu liczby p od n dostaniemy liczbę dość małą i teraz już szybko znajdziemy rozkład tej liczby na sumę różnych liczb pierwszych. Warto jednak odnotować fakt, że znane są długie ciągi kolejnych liczb złożonych, tzw. luki. Luka długości k oznacza, że po liczbie pierwszej p, liczby $p+1, \ldots, p+k-1$ są złożone i dopiero liczba $p+k$ jest pierwsza. Oczywiście długości luk muszą być liczbami parzystymi. Znane są luki do długości 778 (dla liczb pierwszych $p < 72635119999997$) i niektóre inne. To oznacza, że poszukiwanie największej liczby pierwszej nie większej od $n - 10$ może potrwać dość długo, tym bardziej, że sprawdzanie, czy dana liczba jest pierwsza, też jest czasochłonne.

Najniżej występującą luką długości 100 jest luka między liczbami pierwszymi 396733 i 396833. Ale istnieją większe luki. Najdłuższa znana ma długość 863:

Po liczbie 6 505 941 701 960 039 występują 863 liczby złożone. Ta liczba jest większa od 2 miliardów, ale na przykład po liczbie 166726367 występują 194 liczby złożone. J. Young i A. Potler opublikowali (First occurrence of prime gaps, *Math. Comp.*, 1989, **52**, 221–224) tablicę takich luk. Niektóre testy zostały dobrane w taki sposób, by poszukiwanie liczby pierwszej p trwało długo; do tego celu została wykorzystana wspomniana tabela luk (użyto luk długości do 300).

Można jednak zauważyć, że znaleziona liczba pierwsza p jest „dobra" dla wielu liczb n; dokładniej dla liczb n spełniających nierówności

$$p + 10 \leq n < 2p.$$

Można teraz znaleźć taki zbiór skończony liczb pierwszych p, by przedziały $[p+10; 2p-1]$ pokrywały cały przedział $[10; 2\,000\,000\,000]$. Jest wiele metod szukania takich zbiorów. Jeden przykład podaliśmy wcześniej. Inny został wykorzystany w programie wzorcowym. Proponujemy Czytelnikowi jako ćwiczenie zastanowienie się, jak ten przykład został znaleziony.

Testy

Do sprawdzenia rozwiązań użyto 11 testów. Test `GOLO.IN` to test z treści zadania. Następne trzy testy to testy poprawnościowe (odpowiednio dla małych, średniej wielkości i dużych liczb). Następne testy sprawdzały szybkość działania programu i wykorzystywały wspomniane wcześniej luki. Te testy różniły się między sobą wielkością liczb i usytuowaniem luki po to, by sprawdzić różne sposoby szukania liczby pierwszej p mniejszej od $n - 10$.

Spokojna komisja

W parlamencie **Demokratycznej Republiki Bajtocji**, *zgodnie z* **Bardzo Ważną Ustawą**, *należy ukonstytuować* **Komisję Poselską do Spraw Spokoju Publicznego**. *Niestety sprawę utrudnia fakt, iż niektórzy posłowie wzajemnie się nie lubią.*

Komisja musi spełniać następujące warunki:

- *każda partia ma dokładnie jednego reprezentanta w Komisji,*
- *jeśli dwaj posłowie się nie lubią, to nie mogą jednocześnie być w Komisji.*

Każda partia ma w parlamencie dokładnie dwóch posłów. Wszyscy posłowie są ponumerowani liczbami od 1 do $2n$. Posłowie o numerach $2i-1$ i $2i$ należą do partii o numerze i.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `spo.in` *liczbę partii oraz pary posłów, którzy się wzajemnie nie lubią,*
- *wyznaczy skład Komisji, lub stwierdzi, że nie da się jej ukonstytuować,*
- *zapisze wynik w pliku tekstowym* `spo.out`.

Wejście

W pierwszym wierszu pliku tekstowego `spo.in` *znajdują się dwie nieujemne liczby całkowite n i m. Liczba n, spełniająca warunki $1 \leq n \leq 8000$, oznacza liczbę partii. Liczba m, spełniająca warunki $0 \leq m \leq 20000$, oznacza liczbę par nielubiących się posłów. W każdym z kolejnych m wierszy zapisana jest para liczb naturalnych a i b, $1 \leq a < b \leq 2n$, oddzielonych pojedynczym odstępem. Oznacza ona, że posłowie o numerach a i b wzajemnie się nie lubią.*

Wyjście

Plik tekstowy `spo.out` *powinien zawierać pojedyncze słowo* `NIE`, *jeśli utworzenie Komisji nie jest możliwe. W przypadku, gdy utworzenie Komisji jest możliwe, plik* `spo.out` *powinien zawierać n liczb całkowitych z przedziału od 1 do $2n$, zapisanych w kolejności rosnącej i oznaczających numery posłów zasiadających w Komisji. Każda z tych liczb powinna zostać zapisana w osobnym wierszu. Jeśli Komisję można utworzyć na wiele sposobów, Twój program może wypisać dowolny z nich.*

Przykład

Dla pliku wejściowego `spo.in`:

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

poprawną odpowiedzią jest plik wyjściowy `spo.out`:

```
1
4
5
```

Rozwiązanie

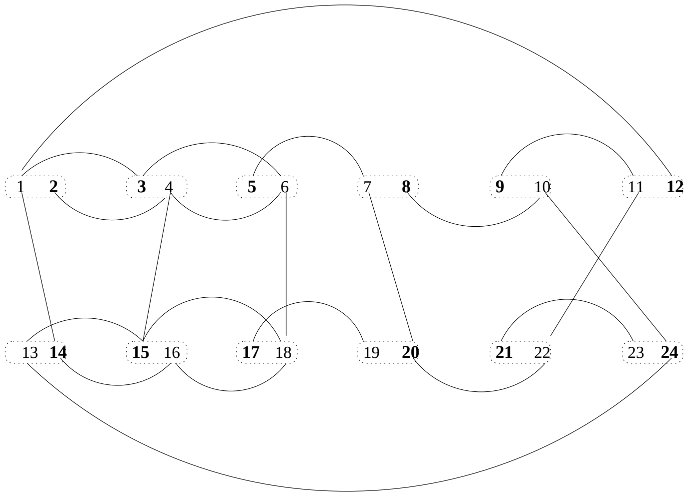
Niech X będzie zbiorem par posłów, którzy się wzajemnie nie lubią. Potraktujmy X jako krawędzie grafu nieskierowanego, a taki graf nazwijmy *grafem konfliktowym*. Przykład grafu konfliktowego ilustruje rysunek 1. Pogrubione numery oznaczają pewien zbiór posłów stanowiący "spokojną" Komisję.

Z grafem konfliktowym stowarzyszymy graf $\mathcal{G}$, który nazwiemy *grafem wymuszeń*. Jeśli x–$y \in X$ oraz $z \neq y$ należy do tej samej partii co y, to $x \to z$ jest skierowaną krawędzią w grafie wymuszeń. Oznaczmy

$$Wymuszone(x) \;=\; \{y \;:\; x \to^* y\},$$

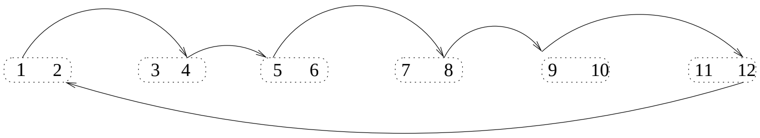
gdzie $\to^*$ oznacza, że w G można przejść z x do y po strzałkach. Zbiór $Wymuszone(x)$ składa się z posłów, których obecność w komisji jest wymuszona przez obecność w niej posła x. Zbiory tego typu mają następującą własność.

Jeśli K jest spokojną Komisją i $x \in K$, to $Wymuszone(x) \subseteq K$.

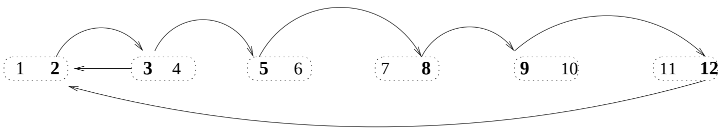


Rys. 1. Przykładowy graf konfliktowy dla 12 partii. Możliwym rozwiązaniem jest zbiór posłów, których numery na rysunku są pogrubione.

Posła x nazwiemy ***kłopotliwym***, jeśli zbiór $Wymuszone(x)$ zawiera dwóch posłów z tej samej partii (co jest niedozwolone). Na rysunku 2 (dla grafu konfliktowego z rysunku 1) poseł 1 jest kłopotliwy, natomiast poseł 2 z rysunku 3 jest niekłopotliwy.



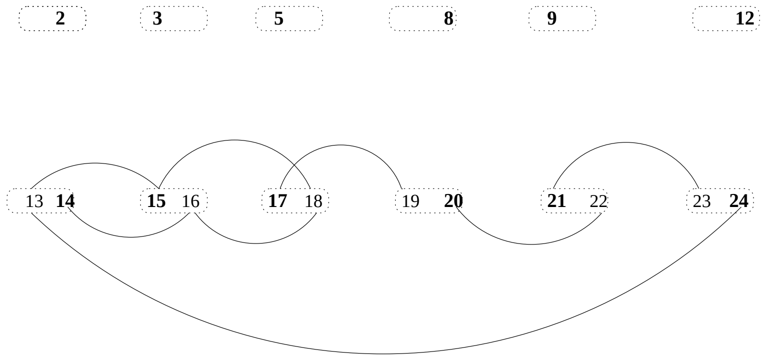
Rys. 2. Część grafu wymuszeń zaczynającego się od 1. Poseł 1 jest kłopotliwy, $\{1,\ 2\} \subseteq Wymuszone(1)$, a 1, 2 są posłami tej samej partii.



Rys. 3. Poseł 2 jest niekłopotliwy: $Wymuszone(2) \;=\; \{2,\ 3,\ 5,\ 8,\ 9,\ 12\}$.

W celu sformowania Komisji wykonujemy następujący algorytm:

```
Algorytm KOMISJA1;
 początkowo zbiorem reprezentantów jest K = ∅;
 while |K| < n do
  begin
   niech π będzie pierwszą partią bez reprezentanta w K;
   if obaj posłowie należący do π są kłopotliwi then
     nie ma rozwiązania, STOP
   else
     begin
      wybierz pierwszego niekłopotliwego posła x ∈ π;
      K := K ∪ Wymuszone(x)
     end
  end
 return K;
```



Rys. 4. Po pierwszej iteracji otrzymujemy reprezentantów pierwszych sześciu partii. Jeśli odrzucimy pozostałych posłów z tych partii, to posłowie z sześciu partii pozostałych są niezależni (w sensie grafu konfliktowego) od wybranych posłów $2, 3, 5, 8, 9, 12$.

Zobaczmy jak działa ten algorytm na naszym przykładzie (Rys. 4). Na początku wybieramy pierwszą partię. Poseł 1 jest kłopotliwy, zatem wybieramy posła 2. $Wymuszone(2) = \{2,\ 3,\ 5,\ 8,\ 9,\ 12\}$. Zbiorem K staje się $\{2,\ 3,\ 5,\ 8,\ 9,\ 12\}$. Następną partią bez reprezentanta jest $\{13,\ 14\}$. Poseł 13 jest kłopotliwy, dodajemy do K zbiór $Wymuszone(14)$. Otrzymujemy końcowy rezultat.

Dowód poprawności.
Algorytm wybiera reprezentanta x z pierwszej partii, a następnie tworzy $K = Wymuszone(x)$. Partie bez reprezentantów w K są, w sensie grafu konfliktowego, całkowicie niezależne od K (zobacz rysunek 4).

Własność niezależności.
Niech $U = Wymuszone(x)$ i niech W będzie sumą teoriomnogościową partii, które są rozłączne z U. Jeśli x jest niekłopotliwy, to w grafie konfliktowym nie ma krawędzi między U i W.

Algorytm znajduje reprezentantów dla pozostałych partii, tak jak gdyby startował od początku. Formalnie poprawność można wykazać indukcyjnie ze względu na liczbę partii.

Zamiast umieszczać w komisji K grupy posłów możemy powiększać komisję po jednym pośle, stosując następującą wersję algorytmu KOMISJA1. Jest to wersja łatwiejsza do zaprogramowania, natomiast poprzednia wersja jest łatwiejsza z punktu widzenia poprawności. W tej wersji dostaniemy dokładnie tę samą komisję.

Oznaczmy i-tą partię przez π_i. Powiemy, że poseł x jest zgodny ze zbiorem posłów w K, gdy x nie jest w konflikcie z żadnym z posłów w K.

Algorytm KOMISJA;
początkowo zbiorem reprezentantów jest $K = \emptyset$;
for $i := 1$ **to** n **do**
begin
wybierz pierwszego niekłopotliwego posła $x \in \pi_i$ zgodnego z K;
if nie ma takiego posła **then** nie ma rozwiązania, STOP;
$K := K \cup \{x\}$
end
return K;

Algorytm Komisja zaimplementowany bezpośrednio może działać zbyt wolno. Sprawdzenie, czy poseł jest niekłopotliwy może wymagać czasu proporcjonalnego do rozmiaru grafu wymuszeń. Zatem łączny czas działania algorytmu wynosi $O(n \cdot (n+m))$. W programie wzorcowym Komisja wybierana jest trochę sprytniej. Rozpoczynamy od znalezienia w grafie wymuszeń wszystkich silnie spójnych składowych, czyli podgrafów, w których od każdego wierzchołka można przejść do każdego innego wierzchołka idąc po strzałkach. Silnie spójne składowe można znaleźć w czasie proporcjonalnym do rozmiaru grafu (zobacz [**?**]). Jeśli istnieje choć jedna silnie spójna składowa zawierająca posłów z tej samej partii, to oczywiście "spokojnej" Komisji nie da się sformować. W przeciwnym przypadku patrzymy na graf silnie spójnych składowych — wierzchołkami są silnie silnie spójne składowe, a od składowej A prowadzi krawędź do B tylko wtedy, gdy w wyjściowym grafie istnieje krawędź od pewnego wierzchołka z A do pewnego wierzchołka z B. Następnie silnie spójne składowe sortujemy topologicznie i rozważamy je w otrzymanej kolejności. Jeśli aktualnie rozważana silnie spójna składowa nie została wcześniej odrzucona, to wybieramy wszystkich należących do niej posłów do Komisji. Dla każdego wybranego w ten sposób posła, odrzucamy silnie spójną składową, która zawiera jego partyjnego kolegę, oraz wszystkie składowe, z których ta składowa jest osiągalna. Jeśli w ten sposób wybierzemy n posłów odpowiadamy TAK, a przeciwnym razie NIE.

Testy

Zadanie testowane było na zestawie 13 danych testowych:

- `spo1.in` — mały test poprawnościowy;
- `spo2.in` — mały test poprawnościowy;
- `spo3.in` — średni test poprawnościowy;
- `spo4a.in` — średni test z odpowiedzią NIE, dla którego "backtracking" działa wykładniczo;
- `spo4b.in` — średni test wydajnościowy, dla którego komisja daje się sformować;
- `spo5a.in` — średni test z odpowiedzią NIE, dla którego "backtracking" działa wykładniczo;
- `spo5b.in` — średni test, dla którego komisja daje się sformować;
- `spo6a.in` — duży test z odpowiedzią NIE, dla którego "backtracking" działa wykładniczo;
- `spo6b.in` — duży test, dla którego komisja istnieje;
- `spo7.in` — test o maksymalnym rozmiarze;
- `spo8a.in` — duży test, dla którego rozwiązanie KOMISJA działa w czasie $O(n^2)$;
- `spo8b.in` — duży test, dla którego rozwiązanie KOMISJA działa w czasie $O(n^2)$ (trochę inny wariant);
- `spo8c.in` — duży test losowy z odpowiedzią TAK.

Testy $4a$ i $4b$, $5a$ i $5b$, $6a$ i $6b$ oraz $8a$, $8b$ i $8c$ były zgrupowane.

Wyspa

Na wyspie o kształcie wielokąta wypukłego o $2n$ bokach, znajduje się $2n-2$ państw — trójkątów, których wierzchołki są jednocześnie wierzchołkami wielokąta. Nie ma państw graniczących dokładnie z dwoma innymi państwami (zatem każde państwo graniczy albo tylko z jednym państwem, albo z trzema). Wynika stąd, że istnieje dokładnie n państw graniczących tylko z jednym państwem (są to państwa **nadmorskie***) oraz $n-2$ państw graniczących z trzema sąsiadami (są to państwa* **wewnątrzlądowe***). Państwa nadmorskie są ponumerowane liczbami od 1 do n, natomiast państwa wewnątrzlądowe mają numery od $n+1$ do $2n-2$. Gdy podróżujemy z jednego państwa do drugiego, to za przekroczenie każdej granicy musimy zapłacić ustaloną stawkę. Poszczególne stawki mogą być różne, ale przekroczenie granicy w obu kierunkach kosztuje tyle samo.*

Dla każdych dwóch państw, spośród n państw nadmorskich, znana jest suma opłat granicznych na drodze prowadzącej (lądem) od jednego państwa do drugiego przez najmniejszą liczbę granic. Zadanie polega na wyznaczeniu wszystkich opłat granicznych na całej wyspie. Dla każdego państwa nadmorskiego należy podać numer państwa, z którym ono graniczy, oraz wysokość odpowiedniej opłaty granicznej. Ponadto, dla każdego z $n-2$ państw wewnątrzlądowych należy podać numery trzech państw, z którymi ono graniczy, oraz wysokości opłat na granicach z tymi państwami.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `wys.in` *sumy opłat granicznych na drogach pomiędzy poszczególnymi państwami nadmorskimi,*
- *obliczy opłaty graniczne na całej wyspie i wyznaczy, które państwa sąsiadują z którymi,*
- *zapisze wyniki w pliku tekstowym* `wys.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `wys.in` *znajduje się jedna dodatnia liczba całkowita n, $4 \le n \le 100$. Jest to liczba państw nadmorskich. W każdym z następnych n wierszy znajduje się n nieujemnych liczb całkowitych oddzielonych pojedynczymi odstępami. Liczba $d_{i,j}$, stojąca na j-tym miejscu w i-tym z tych wierszy, jest równa sumie opłat granicznych na drodze prowadzącej (lądem, przez najmniejszą liczbę granic) z państwa o numerze i do państwa o numerze j. Zakładamy przy tym, że na każdej granicy opłata graniczna jest liczbą całkowitą z przedziału $[1 \ldots 100]$. Oczywiście, $d_{i,j} = d_{j,i}$ oraz $d_{i,i} = 0$.*

Wyjście

W pierwszych n wierszach pliku tekstowego `wys.out` *należy zapisać po dwie liczby całkowite oddzielone pojedynczym ostępem. Pierwszą liczbą w i-tym wierszu powinien być numer państwa graniczącego z państwem o numerze i, a drugą wysokość opłaty granicznej na granicy między tymi dwoma państwami. W każdym z następnych $n-2$ wierszy należy zapisać po sześć liczb oddzielonych pojedynczymi odstępami. W wierszu o numerze i (licząc od początku pliku, a więc $i > n$) pierwszą liczbą ma być numer pierwszego państwa graniczącego z państwem o numerze i, drugą ma być wysokość opłaty granicznej na tej granicy, trzecią ma być numer drugiego państwa graniczącego z państwem o numerze i, czwartą wysokość opłaty granicznej na drugiej granicy, piątą ma być numer trzeciego państwa graniczącego z państwem o numerze i, szóstą ma być wysokość opłaty granicznej na trzeciej granicy. Państwa wewnątrzlądowe mogą być ponumerowane dowolnie liczbami od $n+1$ do $2n-2$.*

Przykład

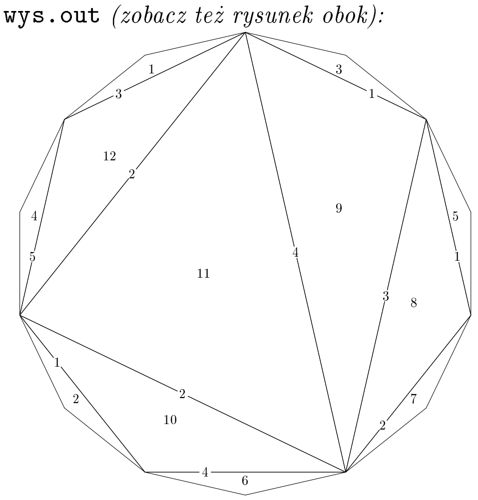
Dla pliku przykładowego `wys.in`*:*

```
7
0 8 10 8 13 11 14
8 0 8 10 11 5 12
```

```
10 8 0 12 5 11 6
8 10 12 0 15 13 16
13 11 5 15 0 14 3
11 5 11 13 14 0 15
14 12 6 16 3 15 0
```

poprawną odpowiedzią jest plik wyjściowy `wys.out` *(zobacz też rysunek obok):*

```
12 3
10 1
9 1
12 5
8 1
10 4
8 2
5 1 7 2 9 3
3 1 8 3 11 4
2 1 6 4 11 2
9 4 10 2 12 2
1 3 4 5 11 2
```



Rozwiązanie

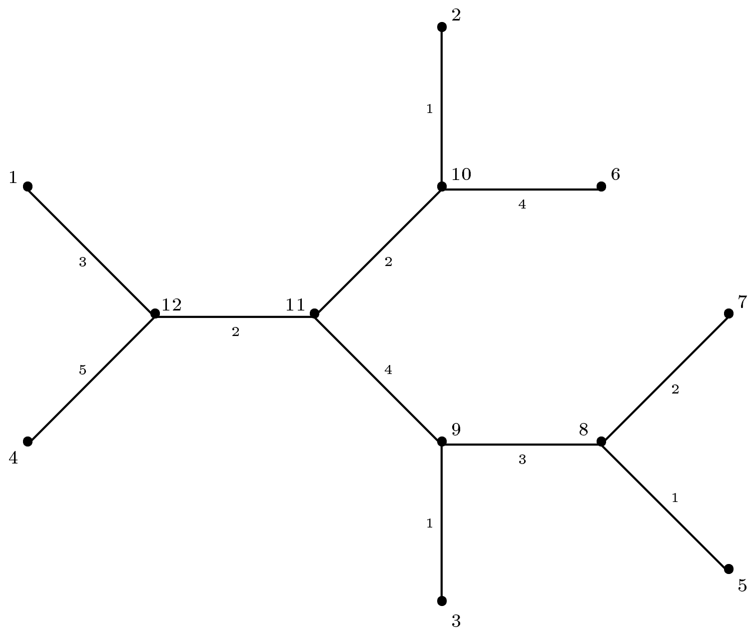
Będziemy używać terminologii teorii grafów — ułatwi to zaprezentowanie możliwych algorytmów prowadzących do rozwiązania zadania. Najprościej mówiąc, grafem nazywamy skończony zbiór punktów na płaszczyźnie (te punkty będziemy zaznaczać na rysunku „grubymi” kropkami) oraz pewien zbiór odcinków łączących te punkty. Wybrane punkty nazywamy **wierzchołkami** grafu, wybrane odcinki **krawędziami** grafu. Te krawędzie mogą się przecinać, będziemy tylko zakładać, że punkt przecięcia dwóch krawędzi nie może być wierzchołkiem grafu. Wierzchołki grafu będziemy numerować początkowymi liczbami naturalnymi, zaczynając od 1.

Krawędziom grafu będziemy przyporządkowywać pewne liczby rzeczywiste. Liczbę rzeczywistą przyporządkowaną krawędzi łączącej wierzchołki k i l będziemy nazywać **długością** tej krawędzi i będziemy oznaczać symbolem $d_{k,l}$.

Liczbę krawędzi wychodzących z jednego wierzchołka nazywamy **stopniem** tego wierzchołka.

W naszym przypadku wierzchołki grafu będą odpowiadać państwom na wyspie. Dwa wierzchołki łączymy krawędzią, jeśli odpowiadające im państwa graniczą ze sobą. Wreszcie długością krawędzi będzie koszt przekraczania granicy. Zauważmy, że w naszym grafie stopień każdego wierzchołka jest równy 1 (dla państw nadmorskich) lub 3 (dla państw wewnątrzlądowych). Nasz graf ma jeszcze jedną ważną własność wynikającą z geometrii wyspy: nie ma w nim cykli. To znaczy, że nie można wyjść z jednego wierzchołka, poruszać się po krawędziach przez inne wierzchołki i powrócić do punktu wyjścia nie przechodząc przez żaden z wierzchołków więcej niż raz. Ponadto z każdego wierzchołka można dojść do każdego innego. Takie grafy nazywamy **drzewami**. Jedną z ważnych własności drzew, wynikającą wprost z tego, że w drzewie nie ma cykli, jest to, że każde dwa wierzchołki łączy tylko jedna droga przebiegająca wzdłuż krawędzi. Długość tej jedynej drogi łączącej wierzchołki k i l (równa sumie długości krawędzi, przez które ta droga przebiega) oznaczymy również symbolem $d_{k,l}$. Jeśli dwa wierzchołki są połączone krawędzią, to oczywiście drogą łączącą je jest ta krawędź, a więc długość tej drogi jest równa długości krawędzi. Zatem użycie tego samego oznaczenia na długość krawędzi i długość drogi nie będzie prowadzić do nieporozumień.

Popatrzmy teraz na przykład. Drzewo odpowiadające wyspie z treści zadania ma postać:



Wierzchołki stopnia 1 nazywamy **liśćmi** drzewa. Pozostałe wierzchołki będziemy nazywać **węzłami**.

Danymi dla zadania są długości dróg łączących liście drzewa. Zadanie polega na tym, by odtworzyć drzewo mając dane wyłącznie odległości między liśćmi. Jest to możliwe dzięki założeniu, że wszystkie węzły mają stopień równy 3. Najprostszy algorytm odtwarzania drzewa polega na znajdowaniu dwóch sąsiednich liści (dwa liście nazywamy *sąsiednimi*, gdy sąsiadują z tym samym węzłem) i zastępowaniu ich węzłem, z którym sąsiadują. Przypuśćmy, że liście o numerach i oraz j sąsiadują z węzłem o numerze k. Usuwamy z tablicy kosztów wiersze i kolumny o numerach i oraz j i dopisujemy nowy wiersz i nową kolumnę o numerze k. W jaki sposób obliczamy długość drogi z węzła k do dowolnego innego liścia m? Zauważmy, że

$$d_{i,m} + d_{j,m} = 2 \cdot d_{k,m} + d_{i,j}.$$

Stąd otrzymujemy wzór

$$d_{k,m} = \frac{1}{2} \cdot (d_{i,m} + d_{j,m} - d_{i,j}).$$

Oczywiście długości krawędzi (i,k) i (j,k) obliczamy natychmiast ze wzorów

$$d_{i,k} = d_{i,m} - d_{k,m}$$

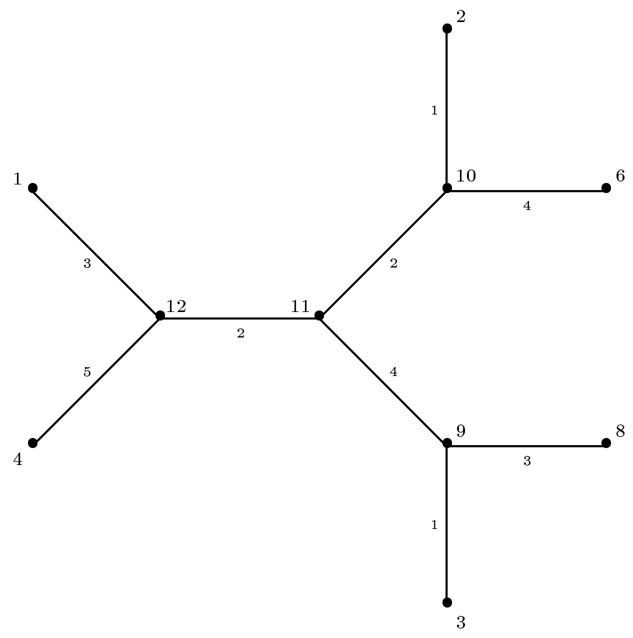
oraz

$$d_{j,k} = d_{j,m} - d_{k,m}.$$

Tak więc, gdy usuwamy dwa liście, tworzymy nowy liść o kolejnym numerze, znajdujemy odległości od tego nowego liścia do pozostałych liści i obliczamy długości usuniętych krawędzi. Wszystkie te dane zapisujemy w odpowiednich tabelach; wypiszemy je po zakończeniu algorytmu. Popatrzmy teraz na tabelę długości dróg.

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	8	10	8	13	11	14
2	8	0	8	10	11	5	12
3	10	8	0	12	4	11	6
4	8	10	12	0	16	13	16
5	13	11	4	16	0	14	3
6	11	5	11	13	14	0	15
7	14	12	6	16	3	15	0

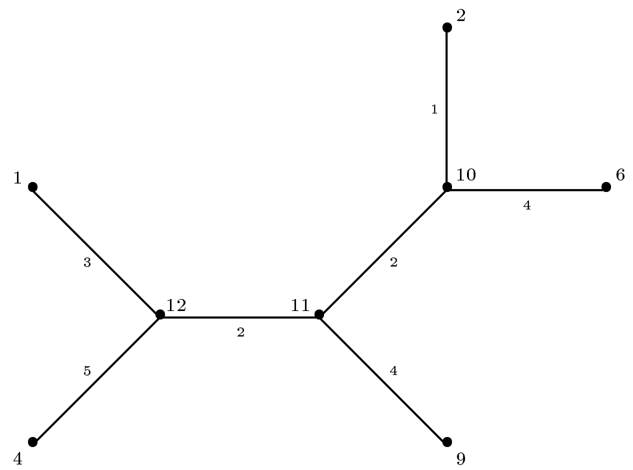
Weźmy sąsiednie liście o numerach 5 i 7. Usuwając je z drzewa otrzymujemy drzewo mniejsze, w którym mamy nowy liść o numerze 8:



Obliczając długości dróg z liścia 8 do pozostałych liści według powyższego wzoru, otrzymamy następującą tabelę długości dróg między liśćmi nowego drzewa:

	1	**2**	**3**	**4**	**6**	**8**
1	0	8	10	8	11	12
2	8	0	8	10	5	10
3	10	8	0	12	11	4
4	8	10	12	0	13	14
6	11	5	11	13	0	13
8	12	10	4	14	13	0

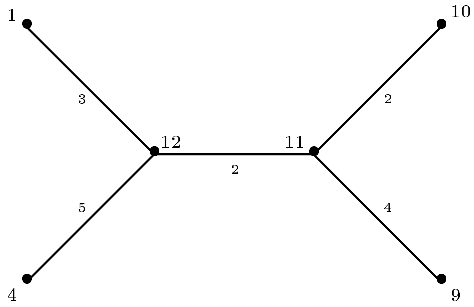
Teraz znajdujemy następne dwa sąsiednie liście: na przykład 3 i 8. Usuwamy je z drzewa i do tablicy długości dróg dopisujemy nowy liść o numerze 9. Nowe drzewo ma postać:



a odpowiednia tabela długości dróg między liśćmi tego drzewa ma postać:

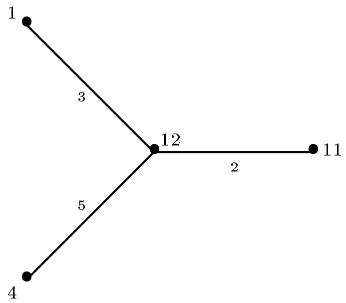
	1	**2**	**4**	**6**	**9**
1	0	8	8	11	9
2	8	0	10	5	7
4	8	10	0	13	11
6	11	5	13	0	10
9	9	7	11	10	0

Kontynuujemy to postępowanie. Znajdujemy kolejne sąsiednie liście: 2 i 6. Usuwamy je z drzewa i z tablicy długości dróg, dopisując nowy liść o numerze 10. Otrzymamy następne drzewo i odpowiadającą mu tablicę długości dróg:



	1	**4**	**9**	**10**
1	0	8	9	7
4	8	0	11	9
9	9	11	0	6
10	7	9	6	0

Jeszcze raz usuwamy dwa sąsiednie liście: tym razem o numerach 9 i 10. Dopisujemy nowy liść o numerze 11. Otrzymamy nowe drzewo i odpowiadającą mu tablicę długości dróg:

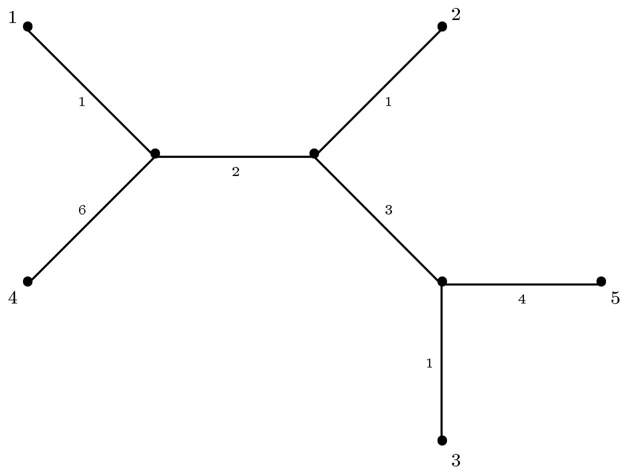


	1	**4**	**11**
1	0	8	5
4	8	0	7
11	5	7	0

Mamy teraz drzewo z trzema liśćmi. Ma ono tylko jeden wierzchołek stopnia 3: dajmy mu numer 12. Oczywiście teraz mamy

$$d_{1,12} = \frac{1}{2} \cdot (d_{1,11} + d_{1,4} - d_{4,11}),$$
$$d_{4,12} = \frac{1}{2} \cdot (d_{1,4} + d_{4,11} - d_{1,11}),$$
$$d_{11,12} = \frac{1}{2} \cdot (d_{4,11} + d_{1,11} - d_{1,4}).$$

W ten sposób całe drzewo zostało odtworzone. Jedynym istotnym problemem w tym algorytmie jest to, w jaki sposób znajdować dwa sąsiednie liście. W powyższym przykładzie za każdym razem wybieraliśmy dwa liście położone najbliżej siebie (tzn. liście i oraz j, dla których liczba $d_{i,j}$ jest najmniejsza). **To udało się tylko dlatego, że drzewo z przykładu zostało specjalnie dobrane!** Ten sposób wybierania sąsiednich liści na ogół jest niepoprawny. Popatrzmy na inny przykład drzewa i odpowiadającej mu tabeli odległości:



	1	**2**	**3**	**4**	**5**
1	0	4	7	7	10
2	4	0	5	9	8
3	7	5	0	12	5
4	7	9	12	0	15
5	10	8	5	15	0

Zauważmy, że najbliżej położone liście (o numerach 1 i 2) nie są sąsiednie! Gdybyśmy mimo to próbowali zastosować poprzedni algorytm do tej sytuacji, otrzymalibyśmy kolejno następujące tabele odległości:

	3	**4**	**5**	**6**
3	0	12	5	4
4	12	0	15	6
5	5	15	0	7
6	4	6	7	0

	4	**5**	**7**
4	0	15	7
5	15	0	4
7	7	4	0

Teraz poprzednie wzory dadzą nam następujące długości krawędzi łączących liście 4, 5 i 7 z wierzchołkiem 8:

$$d_{4,8} = 9, \quad d_{5,8} = 6 \quad \text{oraz} \quad d_{7,8} = -2.$$

Algorytm zakończył się niepowodzeniem, gdyż długość krawędzi nie może być liczbą ujemną.

Istnieją dwie metody znajdowania sąsiednich liści. Jedna z nich polega na obliczeniu dla każdego liścia i wielkości

$$r_i = \sum_j d_{i,j},$$

gdzie sumowanie jest rozciągnięte na wszystkie liście. Następnie dla każdej pary liści i oraz j obliczamy

$$D_{i,j} = d_{i,j} - \frac{r_i + r_j}{n-2}.$$

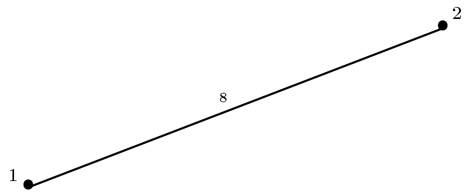
Można wtedy udowodnić twierdzenie mówiące, że liście i oraz j, dla których wartość $D_{i,j}$ jest najmniejsza, są sąsiednie.

Twierdzenie to jest dość sztuczne i nieoczekiwane. Potrzebny jest sposób prostszy i bardziej naturalny. Wybieramy dowolny liść, na przykład o najmniejszym numerze. Niech będzie to numer k. Szukamy liścia sąsiadującego z liściem o numerze l. W tym celu dla każdego liścia m wyznaczamy długość wspólnego odcinka dróg z k do l i z k do m. Jeśli te drogi rozchodzą się w węźle p, to

$$d_{k,p} = \frac{1}{2} \cdot (d_{k,l} + d_{k,m} - d_{l,m}).$$

Możliwe są teraz dwa przypadki. Jeśli liście k i l sąsiadują ze sobą, to węzeł p wypada zawsze w tym samym miejscu i to można łatwo stwierdzić. Jeśli liście k i l nie sąsiadują ze sobą, to znaleziona długość $d_{k,p}$ jest największa wtedy, gdy liść m sąsiaduje z liściem l. W obu przypadkach znajdujemy parę liści sąsiednich: najczęściej będzie to liść l i pewien liść m, może się też zdarzyć, że będą to liście k i l. Poszukiwanie takiej pary liści sąsiednich wymaga czasu $O(n)$, zatem cały algorytm działa w czasie $O(n^2)$.

Inny algorytm polega na sukcesywnym budowaniu drzewa. Zaczynamy od drzewa złożonego tylko z liści o numerach 1 i 2 i krawędzi łączącej te liście. Ma ona długość $d_{1,2}$. W naszym przykładzie to drzewo ma postać:



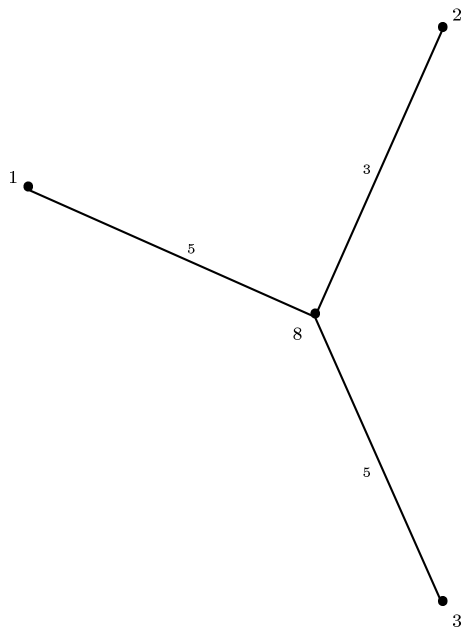
Teraz dołączamy następny liść. Na drodze z liścia 1 do liścia 2 znajduje się węzeł (nadamy mu numer 8), w którym odgałęzia się droga do liścia 3. Można łatwo wyznaczyć długość odcinka od liścia 3 do węzła 8:

$$d_{1,3} + d_{2,3} = 2 \cdot d_{8,3} + d_{1,2}.$$

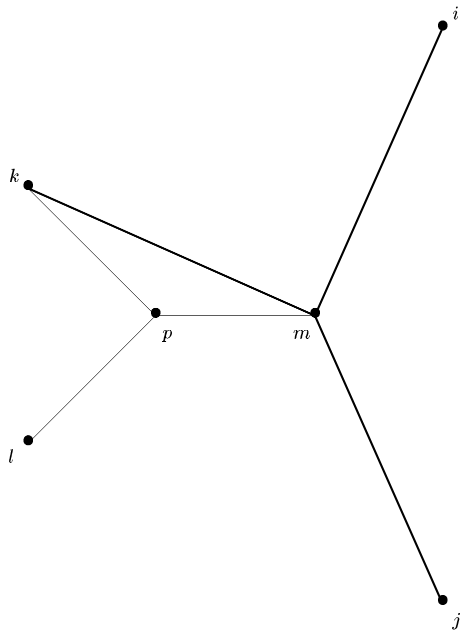
Stąd dostajemy $d_{8,3} = 5$ i następnie

$$d_{1,8} = d_{1,3} - d_{3,8} = 5, \quad d_{2,8} = d_{2,3} - d_{3,8} = 3.$$

Możemy teraz narysować drzewo:



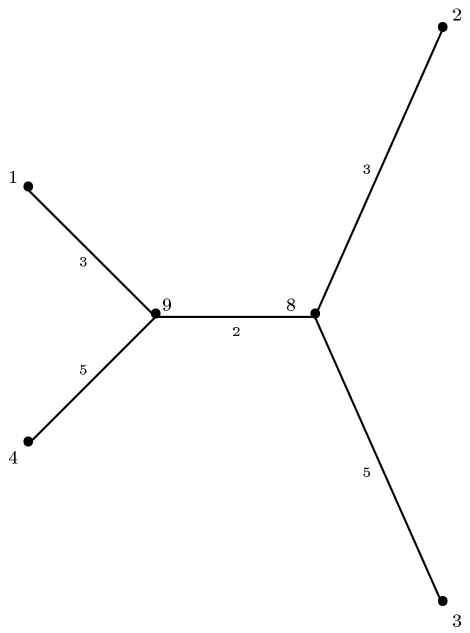
Teraz dołączamy kolejny liść, o numerze 4, wraz z węzłem sąsiadującym z nim. Mamy trzy możliwości. Droga prowadząca z liścia 4 do któregokolwiek z pozostałych liści musi „dołączyć” się do jednej z dróg: z liścia 1 do węzła 8, z liścia 2 do węzła 8 lub z liścia 3 do węzła 8. Oznaczmy liście i węzły literami: węzeł 8 oznaczmy literą m, liść 4 oznaczmy literą l, literą k oznaczmy ten z liści 1,2 lub 3, dla którego droga z k do l nie przechodzi przez m. Pozostałe dwa liście oznaczymy literami i oraz j. Wreszcie literą p oznaczymy węzeł na drodze z k do m, w którym odgałęzia się droga do l. Tę sytuację możemy przedstawić na rysunku:



Na rysunku grubszą linią zaznaczono krawędzie drzewa przed dołączeniem liścia l i węzła p, cieńszymi liniami zaznaczono krawędzie drzewa po dołączeniu tych nowych wierzchołków. Zauważmy, że musi być spełniona nierówność

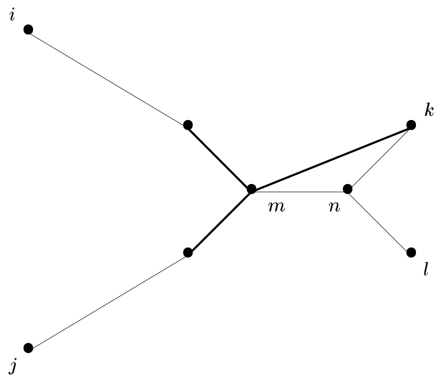
$$d_{i,k} + d_{j,l} > d_{i,j} + d_{k,l}.$$

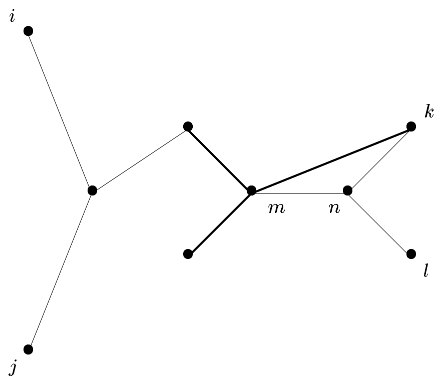
Sprawdzamy zatem tę nierówność dla trzech wyborów k: 1, 2 lub 3. Jedna z tych nierówności będzie spełniona; pokaże ona, w którym miejscu należy dołączyć do drzewa liść l i węzeł p. W naszym przykładzie okaże się, że liść k będzie liściem 1 i węzeł p dołączymy na drodze z liścia 1 do liścia 8. Węzłowi p nadamy teraz kolejny numer, tzn. 9. Tak samo jak poprzednio wyznaczamy odległości między węzłem 9 i wierzchołkami z nim sąsiadującymi. Otrzymamy nowe drzewo:



Teraz musimy dołączyć do drzewa następny liść, o numerze 5. Możliwe są dwa przypadki: droga od liścia 5 do zbudowanego dotychczas drzewa może „dołączyć się" do krawędzi łączącej liść z sąsiednim węzłem lub do krawędzi łączącej dwa węzły. Można łatwo stwierdzić, czy ta droga dołączyła się do krawędzi wychodzącej z liścia.

Przypuśćmy, że mamy dane drzewo i dołączamy nowy liść l. Załóżmy, że droga z liścia l do drzewa dołącza się do krawędzi łączącej liść k z sąsiednim węzłem m. Oznaczmy literą n węzeł, w którym ta droga dołącza się do krawędzi łączącej k z m. Niech wreszcie i i j będą dwoma dowolnymi liśćmi naszego drzewa, różnymi od liścia k. Na następnych dwóch rysunkach widzimy możliwe położenia tych liści i węzłów (grubą linią są zaznaczone krawędzie drzewa, cienką linią drogi w drzewie i nowe krawędzie po dołączeniu liścia l i węzła n).





Sytuacja ta jest możliwa tylko wtedy, gdy zachodzi nierówność

$$d_{i,k} + d_{j,l} > d_{i,j} + d_{k,l}.$$

Ten warunek można sprawdzić dla każdej krawędzi łączącej liść drzewa z sąsiednim węzłem. Może się jednak okazać, że żadna z tych krawędzi nie jest właściwa, czyli powyższa nierówność nie jest prawdziwa. Nowy liść musimy wtedy dołączyć do drzewa w krawędzi łączącej dwa węzły. Jak stwierdzić, która to krawędź?

Można postąpić dwojako. Przypuśćmy, że mamy dołączyć nowy liść l w węźle n znajdującym się na krawędzi łączącej dwa węzły m i k. Niech i i j będą dwoma pozostałymi sąsiadami węzła m. Możemy obliczyć odległości od liścia l do wierzchołków i, j i k (ćwiczeniem dla Czytelnika będzie, jak to zrobić). Dalej postępujemy tak samo: musi być spełniona nierówność

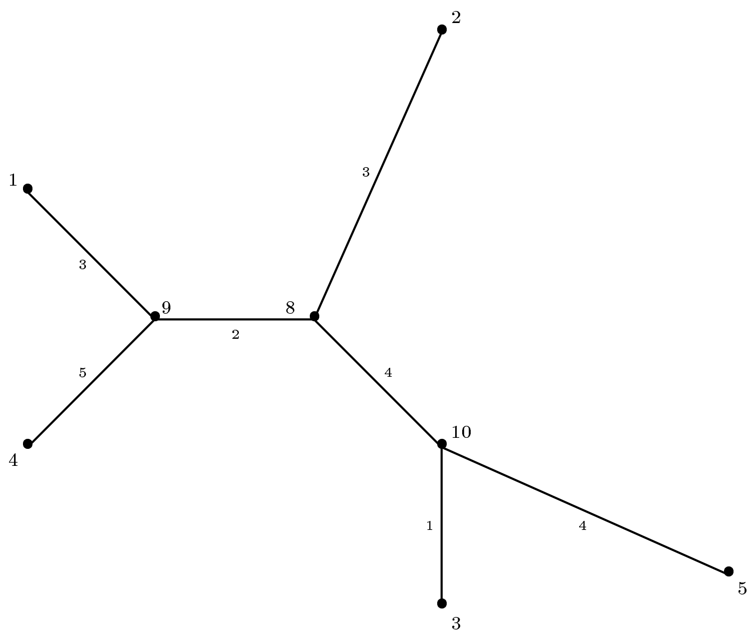
$$d_{i,k} + d_{j,l} > d_{i,j} + d_{k,l}.$$

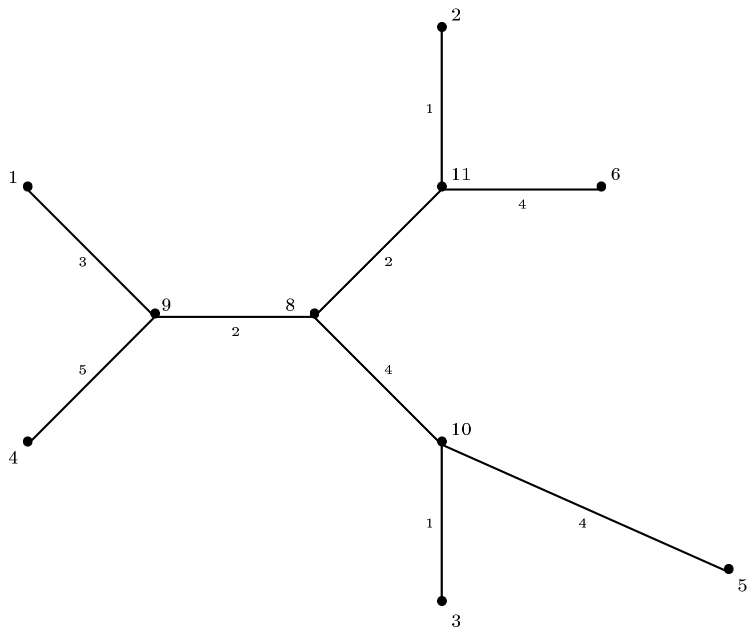
Można też postąpić inaczej. Na czas poszukiwania właściwej krawędzi usuwamy z drzewa krawędzie niewłaściwe. Dokładniej, przeszukujemy krawędzie łączące liście z sąsiednimi węzłami. Jeśli znajdziemy krawędź właściwą, to dołączamy nowy liść. Jeśli badana krawędź nie jest właściwa, to usuwamy ją wraz z liściem, którym ona się kończy. Jej drugi koniec może stać się teraz nowym liściem (zauważmy, że mogą ulec zmianie stopnie węzłów!). Wreszcie znajdziemy krawędź właściwą i wtedy dołączamy z powrotem krawędzie usunięte z drzewa. Ta metoda została użyta w programie wzorcowym.

Oczywiście długości krawędzi obliczamy korzystając z tych samych wzorów co poprzednio.

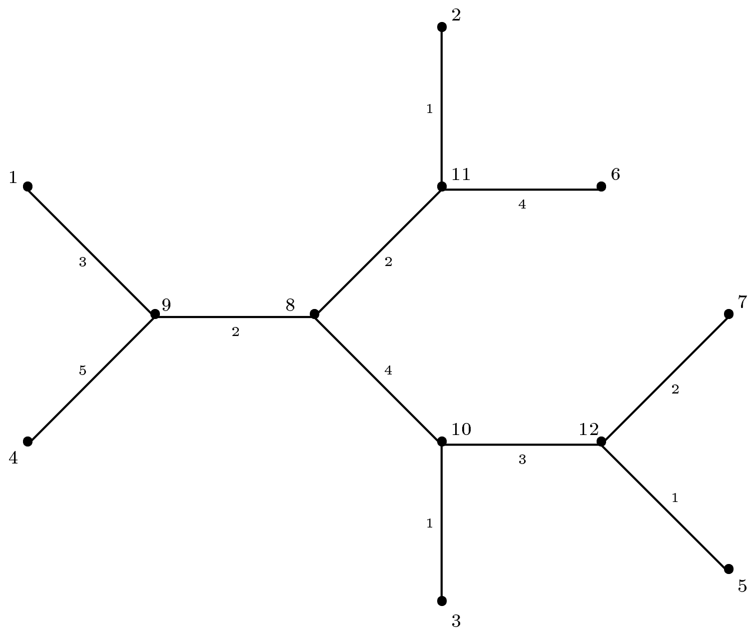
W ten sposób dołączamy kolejne liście i po dołączeniu ostatniego drzewo zostało zrekonstruowane. Czas działania tego algorytmu jest również rzędu $O(n^2)$.

W naszym przykładzie kolejne liście zostaną dołączone do krawędzi łączących inne liście z sąsiednimi węzłami. Na dalszych rysunkach widzimy kolejne drzewa powstające przez dołączenie następnych liści.





I wreszcie drzewo, o które chodziło:



Zadanie to ma zastosowanie w genetyce do wyznaczania tzw. drzew filogenetycznych. Są to drzewa, których liśćmi są gatunki zwierząt lub roślin, a odległości wskazują, jak bardzo dane gatunki są odległe od siebie ewolucyjnie. Rekonstrukcja drzewa sugeruje możliwe kierunki ewolucji: wskazuje, które gatunki mogły powstać bezpośrednio z innych (są to gatunki połączone krawędzią).

Testy

Testy zostały stworzone automatycznie, charakteryzuje je 5 wartości, przy tworzeniu drzewa binarnego:

- n,
- p_{min} — minimalny procent liści, jaki będzie miało lewe poddrzewo,
- p_{max} — maksymalny procent,
- o_{min} — minimalna opłata graniczna,

- o_{max} — maksymalna opłata.

Jeśli procent liści jest bliski 50, to otrzymamy drzewo pełne. Jeśli bliski 0, będzie to lista. Jeśli będzie dowolny od 0 do 100 to otrzymamy drzewo losowe, czyli zrównoważone.

test	n	p_{min}	p_{max}	o_{min}	o_{max}
wys1.in	6	0	100	1	7
wys2.in	10	0	100	1	7
wys3.in	15	0	100	1	1
wys4.in	15	0	5	1	10
wys5.in	15	45	55	1	10
wys6.in	20	0	100	1	20
wys7.in	40	0	100	1	50
wys8.in	40	0	5	1	50
wys9.in	70	45	45	1	1
wys10.in	100	50	50	1	100
wys11.in	100	0	5	100	100
wys12.in	100	0	100	1	100

Krzysztof Loryś
Treść zadania, Opracowanie

Tomasz Waleń
Program

Mrówki i biedronka

Jak wiadomo, mrówki potrafią "hodować" mszyce. Mszyce wydzielają słodką rosę miodową, którą spijają mrówki. Mrówki zaś bronią mszyc przed ich największymi wrogami — biedronkami.

Na drzewie obok mrowiska znajduje się właśnie taka hodowla mszyc. Mszyce żerują na liściach oraz w rozgałęzieniach drzewa. W niektórych z tych miejsc znajdują się również mrówki patrolujące drzewo. Dla ustalenia uwagi, mrówki są ponumerowane od jeden w górę. Hodowli zagraża biedronka, która zawsze siada na drzewie tam, gdzie są mszyce, czyli na liściach lub w rozgałęzieniach. W chwili, gdy gdzieś na drzewie usiądzie biedronka, mrówki patrolujące drzewo ruszają w jej stronę, aby ją przegonić. Kierują się przy tym następującymi zasadami:

- *z każdego miejsca na drzewie (liścia lub rozgałęzienia) można dojść do każdego innego miejsca (bez zawracania) tylko na jeden sposób; każda mrówka wybiera właśnie taką drogę do miejsca lądowania biedronki,*
- *jeżeli w miejscu lądowania biedronki znajduje się mrówka, biedronka natychmiast odlatuje,*
- *jeżeli na drodze, od aktualnego położenia mrówki do miejsca lądowania biedronki, znajdzie się inna mrówka, to ta położona dalej od biedronki kończy wędrówkę i zostaje w miejscu swojego aktualnego położenia,*
- *jeżeli dwie lub więcej mrówek próbuje wejść na to samo rozgałęzienie drzewa, to robi to tylko jedna mrówka — ta z najmniejszym numerem, a reszta mrówek pozostaje na swoich miejscach (liściach lub rozgałęzieniach),*
- *mrówka, która dociera do miejsca lądowania biedronki, przegania ją i pozostaje w tym miejscu.*

Biedronka jest uparta i znowu ląduje na drzewie. Wówczas mrówki ponownie ruszają, aby przegonić intruza.

Dla uproszczenia przyjmujemy, że przejście gałązki łączącej liść z rozgałęzieniem lub łączącej dwa rozgałęzienia, zajmuje wszystkim mrówkom jednostkę czasu.

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `mro.in` *opis drzewa, początkowe położenia mrówek oraz miejsca, w których kolejno siada biedronka,*
- *dla każdej mrówki znajdzie jej końcowe położenie i wyznaczy liczbę mówiącą, ile razy przegoniła ona biedronkę.*

Wejście

W pierwszym wierszu pliku tekstowego `mro.in` *znajduje się jedna liczba całkowita n, równa łącznej liczbie liści i rozgałęzień w drzewie, $1 \leq n \leq 5000$. Przyjmujemy, że liście i rozgałęzienia są ponumerowane od 1 do n. W kolejnych $n-1$ wierszach są opisane gałązki — w każdym z tych wierszy są zapisane dwie liczby całkowite a i b oznaczające, że dana gałązka łączy miejsca a i b. Gałązki pozwalają na przejście z każdego miejsca na drzewie, do każdego innego miejsca. W $n+1$-szym wierszu jest zapisana jedna liczba całkowita k, $1 \leq k \leq 1000$ i $k \leq n$, równa liczbie mrówek patrolujących drzewo. W każdym z kolejnych k wierszy zapisana jest jedna liczba całkowita z przedziału od 1 do n. Liczba zapisana w wierszu $n+1+i$ oznacza początkowe położenie mrówki nr i. W każdym miejscu (liściu lub rozgałęzieniu) może znajdować się co najwyżej jedna mrówka. W wierszu $n+k+2$ zapisana jest jedna liczba całkowita l, $1 \leq l \leq 500$, mówiąca ile razy biedronka siada na drzewie. W każdym z kolejnych l wierszy zapisana jest jedna liczba całkowita z zakresu od 1 do n. Liczby te opisują kolejne miejsca, w których siada biedronka.*

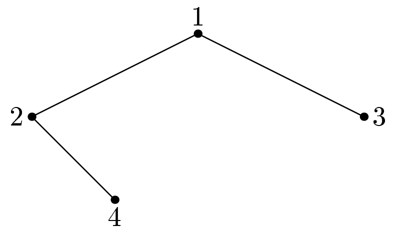
Wyjście

Twój program powinien zapisać k wierszy w pliku wyjściowym `mro.out`. *W i-tym wierszu powinny zostać zapisane dwie liczby całkowite oddzielone pojedynczym odstępem — końcowa pozycja i-tej mrówki (numer rozgałęzienia lub liścia) i liczba mówiąca, ile razy przegoniła ona biedronkę.*

Przykład

Dla pliku wejściowego `mro.in`*:*

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



Rysunek 1. Drzewo dla przykładowych danych

poprawną odpwiedzią jest plik wyjściowy `mro.out`*:*

```
1 0
4 2
```

Rozwiązanie

Narzucający się schemat rozwiązania polega na symulacji kolejnych zdarzeń (lądowań biedronki i pogoni mrówek). Po wylądowaniu biedronki w wierzchołku drzewa (tj. liściu lub rozgałęzieniu) symulujemy ruchy mrówek. Oznaczmy przez B wierzchołek, w którym wylądowała biedronka. Kolejno dla każdej mrówki (wg rosnących numerów) sprawdzamy czy droga między nią a wierzchołkiem B jest wolna, tj. czy w żadnym wierzchołku na tej drodze nie znajduje się jakaś inna mrówka. Jeśli droga jest wolna, przesuwamy mrówkę do następnego wierzchołka.

Pierwszy problem z jakim musimy sobie poradzić, to sposób wyznaczania drogi między wierzchołkami zajmowanymi przez mrówki a wierzchołkiem B. W tym celu musimy przyjąć odpowiednią reprezentację drzewa w pamięci. Wygodnie jest przyjąć, że dla każdego wierzchołka drzewa pamiętamy listę jego sąsiadów oraz wskaźnik na ojca. Początkowo jako korzeń możemy obrać dowolny wierzchołek drzewa a następnie, po każdorazowym lądowaniu biedronki, modyfikować to drzewo tak, by korzeniem zostawał wierzchołek B. Zauważmy, że modyfikacja ta ograniczać się będzie do odwrócenia wskaźników pomiędzy sąsiednimi wierzchołkami, leżącymi na ścieżce od poprzedniego korzenia do wierzchołka B. Teraz dla każdej mrówki droga do wierzchołka B wyznaczona jest poprzez wskaźniki na ojca. Oczywistym jest, że złożoność czasowa tej fazy algorytmu jest ograniczona przez $O(n)$, gdzie n jest liczbą wierzchołków w drzewie.

Drugi problem polega na efektywnym sprawdzaniu czy droga między wierzchołkiem, w którym znajduje się mrówka a wierzchołkiem B jest wolna. Jedno z rozwiązań polega na tym, by dla każdego wierzchołka pamiętać znacznik, mówiący czy dany wierzchołek leży na drodze wolnej. Początkowego ustawienia znaczników można dokonać prostą procedurą przechodzenia drzewa, np. procedurą przechodzenia w głąb. Każdy ruch mrówki może powodować konieczność "zablokowania" znaczników niektórych wierzchołków. Jeśli mrówka przesunęła się do wierzchołka v, to należy zablokować znaczniki wierzchołków leżących w poddrzewie o korzeniu v. Oczywiście każdy znacznik będzie zablokowany co najwyżej jeden raz, a więc i ta faza algorytmu ma złożoność czasową ograniczoną przez $O(n)$.

W rozwiązaniu wzorcowym przyjęto inną metodę rozwiązania problemu wolnych dróg. Wygląda ona na nieco bardziej skomplikowaną, jednak pozwala na bardziej efektywną implementację algorytmu, zwłaszcza w sytuacji gdy liczba mrówek m jest istotnie mniejsza od n. Symulując ruchy mrówek zapamiętujemy w wierzchołkach ich "ślady" (numer mrówki i numer kroku). Każda mrówka przesuwana jest do przodu dopóty, dopóki nie napotka wierzchołka z zapamiętanym śladem innej mrówki, bądź też któraś z mrówek nie dojdzie do wierzchołka B. Informacje zgromadzone w czasie symulacji pozwalają na wyliczenie właściwych odległości, na jakie powinny się przemieścić poszczególne mrówki. Zauważmy, że odległości te można by w prosty sposób obliczyć przechodząc drzewo w głąb: dla każdej mrówki jest to najmniejszy numer kroku zapamiętany w śladzie w wierzchołku znajdującym się na drodze tej mrówki do B. Taka metoda nie byłaby jednak specjalnie oszczędna. Jak łatwo zauważyć wszystkie istotne dla tych obliczeń informacje znajdują się w śladach zapamiętanych w tych wierzchołkach, w których doszło do "spotkań" mrówek ze śladami innych mrówek. Można więc w trakcie symulacji zbudować graf (a w istocie drzewo) spotkań i następnie zastosować do tego grafu procedurę przechodzenia w głąb. Ponieważ graf ten ma $O(m)$ wierzchołków, taką też złożoność ma procedura przechodzenia tego grafu.

Może się wydawać, że nie unikniemy jednak przechodzenia całego drzewa (a więc kosztu zależnego od n) w drugiej fazie algorytmu, ponieważ musimy pozacierać ślady przed symulacją kolejnego lądowania biedronki. To fakt. Możemy jednak pamiętać ślady nie w wierzchołkach drzewa, lecz w tablicy indeksowanej numerami wierzchołków. Zacieranie śladów możemy wówczas wykonać, stosując efektywne procedury zerowania tablicy.

Testy

Do testowania rozwiązania tego zadania użyto zestawu 15 testów:

- `mro1.in` — mały test poprawnościowy;
- `mro1a.in` — test poprawnościowy, mrówki we wszystkich węzłach;
- `mro1b.in` — test poprawnościowy, jedna mrówka;
- `mro2.in` — test poprawnościowy;
- `mro2a.in` — test poprawnościowy, drzewo z jednym węzłem;
- `mro2b.in` — test poprawnościowy;
- `mro3.in` — n=100, m=10, drzewo binarne;
- `mro4.in` — n=500, m=10, drzewo losowe;
- `mro5.in` — n=1000, m=10, drzewo binarne;
- `mro6.in` — n=3000, m=3, mrówki na gwieździe;
- `mro7.in` — n=5000, m=1000, mrówki na gwieździe;
- `mro8.in` — n=5000, drabinka, dwie biedronki;
- `mro9.in` — n=5000, dwie mrówki na liście z losowymi wypustkami, losowa biedronka;
- `mro10.in` — n=5000, dwie mrówki na liście;
- `mro11.in` — n=5000, m=3, mrówki na gwieździe.

Zbigniew J. Czech
Treść zadania, Opracowanie

Marcin Stefaniak
Program

Podróż

Rozważmy graf opisujący sieć komunikacji publicznej, np. sieć autobusową, tramwajową, metra, itp. Wierzchołki grafu, o numerach 1, 2, ..., n, reprezentują przystanki, a krawędzie $\langle p_i,\ p_j\rangle$, *(dla* $p_i \neq p_j$*) oznaczają możliwe, bezpośrednie przejazdy od przystanku* p_i *do* p_j $(1 \le p_i, p_j \le n)$.

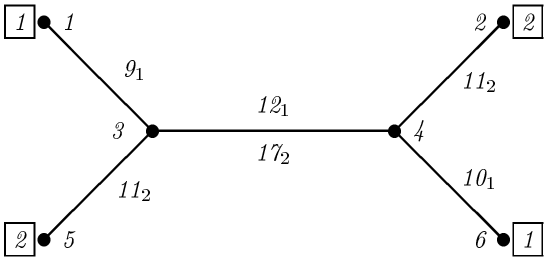
W sieci kursują pojazdy linii komunikacyjnych. Linie komunikacyjne oznaczone są numerami 1, 2, ..., k. Linia komunikacyjna l zdefiniowana jest przez ciąg przystanków $p_{l,1}, p_{l,2}, \ldots, p_{l,s_l}$, *przez które przejeżdżają pojazdy linii, oraz czasy przejazdów* $r_{l,1}, r_{l,2}, \ldots, r_{l,s_l-1}$ *pomiędzy przystankami —* $r_{l,1}$ *jest czasem przejazdu od przystanku* $p_{l,1}$ *do* $p_{l,2}$, *lub z powrotem, tj. od przystanku* $p_{l,2}$ *do* $p_{l,1}$; $r_{l,2}$ *jest czasem przejazdu od przystanku* $p_{l,2}$ *do* $p_{l,3}$, *itd. Wszystkie przystanki linii są różne, tj. dla* $i \neq j$ *zachodzi* $p_{l,i} \neq p_{l,j}$.

Na danej linii l pojazdy kursują z określoną częstotliwością c_l, *gdzie* c_l *jest liczbą ze zbioru* $\{6, 10, 12, 15, 20, 30, 60\}$. *Pojazdy linii wyruszają z przystanku* $p_{l,1}$ *o każdej "okrągłej" godzinie doby, g:0,* $(0 \le g \le 23)$, *a następnie zgodnie z częstotliwością linii, a więc o godzinach* $g{:}c_l$, $g{:}2c_l$, *... itd. (*$g{:}c_l$ *oznacza `*c_l *minut po godzinie g"). Ruch pojazdów linii odbywa się jednocześnie w obu kierunkach: z przystanku* $p_{l,1}$ *do* p_{l,s_l}, *a także z przystanku* p_{l,s_l} *do* $p_{l,1}$. *Godziny odjazdów pojazdów linii z przystanku* p_{l,s_l} *są takie same, jak z przystanku* $p_{l,1}$.

W tak zdefiniowanej sieci komunikacji publicznej chcemy odbyć podróż z przystanku początkowego x, do przystanku końcowego y. Zakładamy, że podróż jest możliwa i nie będzie trwała dłużej niż 24 godziny. W trakcie podróży możemy się przesiadać dowolną liczbę razy z jednej linii komunikacyjnej na inną. Przyjmujemy, że czas dokonania przesiadki jest równy 0, jednakowoż, zmieniając linię musimy liczyć się z koniecznością czekania na pojazd linii, do którego chcemy się przesiąść. Naszym celem jest odbycie podróży z przystanku początkowego x, do przystanku końcowego y, w jak najkrótszym czasie.

Przykład

Na poniższym rysunku przedstawiono schemat sieci komunikacyjnej o 6 przystankach i dwóch liniach: $\boxed{1}$ *i* $\boxed{2}$. *Pojazdy linii* $\boxed{1}$ *kursują pomiędzy przystankami 1, 3, 4 i 6, a linii* $\boxed{2}$ *pomiędzy przystankami 2, 4, 3 i 5. Częstotliwości kursowania pojazdów linii wynoszą, odpowiednio,* $c_1 = 15$ *oraz* $c_2 = 20$. *Czasy przejazdów pomiędzy przystankami umieszczono obok krawędzi sieci opatrując je indeksami 1 i 2 dla poszczególnych linii.*



Załóżmy, że o godzinie 23:30 znajdujemy się na przystanku początkowym 5 i chcemy odbyć podróż do przystanku końcowego 6. Wówczas musimy odczekać 10 minut i o godzinie 23:40 wyjeżdżamy linią $\boxed{2}$. *Nasza podróż może mieć dwa warianty. W pierwszym wariancie, po dojechaniu do przystanku 3 o godzinie 23:51 i odczekaniu 3 minut, przesiadamy się o godzinie 23:54 na linię* $\boxed{1}$ *i dojeżdżamy do przystanku 6 o godzinie 0:16 (następnego dnia). W drugim wariancie, dojeżdżamy linią* $\boxed{2}$ *do przystanku 4 o godzinie 0:8, czekamy 13 minut i o godzinie 0:21 wsiadamy do pojazdu linii* $\boxed{1}$, *osiągając przystanek końcowy 6 o godzinie 0:31. Tak więc najwcześniej możemy dotrzeć do przystanku 6 o godzinie 0:16.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `pod.in` *opis sieci oraz linii komunikacyjnych, numer przystanku początkowego x, numer przystanku końcowego y, godzinę początkową* g_x *oraz minuty początkowe* m_x,
- *wyznaczy minimalny czas podróży od przystanku początkowego x, do przystanku końcowego y,*
- *zapisze do pliku tekstowego* `pod.out` *najwcześniejszą możliwą godzinę z minutami dotarcia do przystanku y — odpowiednio,* g_y *oraz* m_y.

Wejście

W pierwszym wierszu pliku tekstowego `pod.in` *zapisanych jest sześć liczb całkowitych, pooddzielanych pojedynczymi odstępami:*

- *liczba przystanków n ($1 \leq n \leq 1000$),*
- *liczba linii komunikacyjnych k ($1 \leq k \leq 2000$),*
- *numer przystanku początkowego x ($1 \leq x \leq n$),*
- *numer przystanku końcowego y ($1 \leq y \leq n$),*
- *godzina rozpoczęcia podróży g_x ($0 \leq g_x \leq 23$),*
- *minuta rozpoczęcia podróży m_x ($0 \leq m_x \leq 59$).*

Przystanki są ponumerowane od 1 do n, a linie komunikacyjne od 1 do k.

W kolejnych $3k$ wierszach opisane są kolejne linie komunikacyjne — opis każdej linii zajmuje trzy kolejne wiersze.

- *W pierwszym wierszu opisującym linię komunikacyjną l są zapisane dwie liczby całkowite, oddzielone pojedynczym odstępem: s_l, równa liczbie przystanków ($2 \leq s_l \leq n$), oraz c_l, równa częstotliwości kursowania pojazdów ($c_l \in \{6, 10, 12, 15, 20, 30, 60\}$).*
- *W drugim wierszu opisującym linię komunikacyjną l jest zapisanych s_l różnych liczb całkowitych, pooddzielanych pojedynczymi odstępami: $p_{l,1}, p_{l,2}, \ldots, p_{l,s_l}$ — numery kolejnych przystanków na tej linii ($1 \leq p_{l,i} \leq n$, dla $1 \leq i \leq s_l$).*
- *W trzecim wierszu opisującym linię komunikacyjną l jest zapisanych $s_l - 1$ liczb całkowitych, pooddzielanych pojedynczymi odstępami: $r_{l,1}, r_{l,2}, \ldots, r_{l,s_l-1}$ — czasy przejazdów (w minutach) pomiędzy kolejnymi przystankami na tej linii ($1 \leq r_{l,i} \leq 240$, dla $1 \leq i \leq s_l - 1$).*

Suma liczb przystanków, na wszystkich liniach razem, nie przekracza 4000 (tzn. $s_1 + s_2 + \ldots + s_k \leq 4000$).

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `pod.out` *dwie liczby całkowite oddzielone pojedynczym odstępem: godzinę dotarcia do przystanku końcowego g_y ($0 \leq g_y \leq 23$) oraz minutę dotarcia do przystanku końcowego m_y ($0 \leq m_y \leq 59$).*

Przykład

Dla pliku wejściowego `pod.in`*:*

```
6 2 5 6 23 30
4 15
1 3 4 6
9 12 10
4 20
5 3 4 2
11 17 11
```

poprawną odpowiedzią jest plik wyjściowy `pod.out`*:*

```
0 16
```

Rozwiązanie

W pierwszej kolejności zaprojektujemy strukturę danych, w której będziemy przechowywać rozkład jazdy pojazdów. Z zadania wynika, że sieć komunikacyjna działa całą dobę. Pojazdy linii komunikacyjnych wyruszają z przystanków krańcowych z określoną częstotliwością, c, gdzie c jest liczbą ze zbioru {6, 10, 12, 15, 20, 30, 60}. Ponieważ każda z tych częstotliwości dzieli liczbę 60, to pojazdy danej linii będą odjeżdżać z przystanków zawsze w tych samych minutach dowolnej godziny. Przykładowo, niech dla pewnej linii $c = 15$. Wówczas pojazdy tej linii wyruszają z przystanku p_1 o godzinie g minut 0, 15, 30, 45, gdzie $g \in \{0, 1, \ldots, 23\}$. Podobnie, czasy odjazdu z przystanku p_2 są równe godzina g minut $(0 + r_1) \bmod 60$, $(15 + r_1) \bmod 60$, $(30 + r_1) \bmod 60$, $(45 + r_1) \bmod 60$, gdzie r_1 jest czasem przejazdu od przystanku p_1 do p_2. Tak więc, zapamiętanie rozkładu jazdy na dowolnym przystanku wymaga przechowania jednej z chwil odjazdów wyrażonej jako liczba minut po (dowolnej) godzinie g (pozostałe chwile odjazdów można łatwo obliczyć znając częstotliwość kursowania linii). Dane określające rozkład jazdy dla dowolnego przystanku pi można przechować w rekordach o postaci:

```
type dane_odj = { dane dotyczące odjazdów z przystanku pi }
  record
     pi:  integer; { pi – przystanek aktualny }
     pj:  integer; { pj – przystanek następny (sąsiedni) }
     r:   byte; { “czysty” czas przejazdu od przystanku }
                { pi do pj }
     go:  byte; { jedna z chwil odjazdów pamiętana jako liczba }
                { minut po (dowolnej) godzinie g }
     c:   byte; { częstotliwość kursowania pojazdów linii, }
                { której dotyczy niniejszy rekord }
  end;
```

zaś cały rozkład jazdy w tablicy:

rozkl_jazdy: **array**[1 .. q] **of** *dane_odj*;

gdzie q jest całkowitą liczbą rekordów. Zauważmy, że dla dowolnego przystanku pi, z wyjątkiem przystanków krańcowych, liczba rekordów w rozkładzie jazdy jest równa podwojonej liczbie linii komunikacyjnych (bo linie prowadzą w obie strony), których pojazdy dojeżdżają, a następnie odjeżdżają z tego przystanku. Dla przystanków krańcowych liczba rekordów jest równa liczbie linii komunikacyjnych wychodzących z tych przystanków. Dla przykładowej sieci z treści zadania, $q = 12$. Liczby rekordów w rozkładzie jazdy dla wierzchołków od 1 do 6 są równe, odpowiednio, 1, 1, 4, 4, 1 oraz 1.

Zadanie można rozwiązać adaptując do tego celu algorytm Dijkstry znajdowania najkrótszych dróg w grafie (zobacz [**?**]). W naszym przypadku czas przejazdu między przystankami nie jest stały i zależy od chwili h, w której przybywamy na przystanek. Znając chwilę h, przystanki pi, pj oraz linię komunikacyjną, którą podróżujemy (tj. odpowiedni rekord w tablicy *rozkl_jazdy*), czas przejazdu pomiędzy przystankami pi oraz pj można wyznaczyć za pomocą funkcji:

```
function czas_przejazdu( pi, pj: integer; h: integer): integer;
begin
   czas_przejazdu := ((60 + pi − h) mod c) + r;
end
```

Wartość $(60 + go - h) \bmod c$ jest czasem oczekiwania na przystanku pi, zaś r jest “czystym” czasem przejazdu wybraną linią od przystanku pi do pj.

Stosując algorytm Dijkstry do rozwiązania naszego zadania będziemy wyróżniać wśród wszystkich przystanków sieci te z nich, dla których najkrótszy czas dojazdu z przystanku początkowego x został już ustalony. Do tego celu wprowadzimy tablicę:

ustal: **array**[1 .. n] **of** *boolean*;

w której wartość $ustal[i] = true$, $i = 1, 2, \ldots, n$, oznacza, że znamy już najkrótszy czas dojazdu od przystanku x do i. Początkowo inicjujemy wszystkie elementy na *false*, za wyjątkiem elementu $ustal[x]$, któremu przypisujemy wartość *true*. Będziemy także korzystać z tablicy:

D: **array**[1 .. n] **of** *integer*;

w której zapamiętamy najkrótsze czasy dojazdu wyrażone w minutach od przystanku x do przystanku i, $i = 1, 2, \ldots, n$, $i \neq x$. Elementy tablicy inicjujemy wartością ∞, za wyjątkiem przystanku x, dla którego przyjmujemy $D[x] = 0$. Oto rozwiązanie naszego zadania:

```
D[x] := 0; { przyjęcie czasu dojazdu do przystanku x jako równego 0 }
for k := 1 to n do
  { W każdej iteracji ustalany jest najkrótszy czas przejazdu }
  { z przystanku x do jednego z przystanków sieci. }
  Znajdź indeks w taki, że D[w] jest minimalne wśród wszystkich
     D[i], i = 1, 2, ..., n, dla których ustal[i] = false;
  if D[w] = ∞ then
```

```
8:      Nie istnieje ścieżka do przystanku końcowego; break;
9:    end if;
10:   ustal[w] := true;
11:   if w = y then
12:      break; { Zadanie rozwiązano; wynik: D[w] = D[y] }
13:   end if;
14:   for wszystkich połączeń (w, i) między sąsiednimi przystankami
15:         w oraz i, dla których ustal[i] = false do
16:      a := czas_przejazdu(w, i, (m_x + D[w]) mod 60);
17:      if (a ≠ ∞) and (D[w] + a < D[i]) then
18:         D[i] := D[w] + a;
19:      end if;
20:   end for;
21: end for;
22: t := g_x * 60 + m_x + D[y];
23: writeln((t div 60) mod 24, ' ', t mod 60);
```

W każdej iteracji instrukcji **for** w wierszach 2–21 zostaje dołączony jeden przystanek, dla którego zostaje ustalony minimalny czas dojazdu z przystanku początkowego x. Przystanek ten, o indeksie w, ma najkrótszy czas dojazdu $D[w]$ przez wszystkie przystanki, dla których najkrótsze czasy zostały już ustalone. W wierszach 14–20 sprawdzamy czy po dołączeniu przystanku w, czasy dojazdu do pozostałych przystanków o nieustalonym czasie dojazdu przez przystanek w nie uległy skróceniu. Jeżeli tak, to czasy te modyfikujemy.

Złożoność czasowa poprawiania tablicy D (wiersze 14–20) jest $O(q)$, gdzie q jest liczbą rekordów w tablicy *rozkl_jazdy*. Złożoność czasowa n-krotnego wyszukiwania minimalnej wartości $D[w]$ jest $O(n^2)$

W efektywnej implementacji algorytmu warto jest pogrupować rekordy tablicy *rozkl_jazdy* według numerów przystanków odjazdu, ponieważ uaktualniając tablicę D (wiersze 14–20) przeglądamy połączenia wychodzące z przystanku w. Numery przystanków są liczbami całkowitymi z zakresu $1..n$, więc grupowanie takie można zrealizować za pomocą sortowania pozycyjnego z użyciem liczników częstości w czasie $O(n+q)$ (zobacz [**?**]). Podsumowując, złożoność czasowa przedstawionego rozwiązania jest $O(n^2+q)$.

Testy

Do generowania losowych testów użyto generatora **gen_pod.pas**, który losuje zestaw danych o zadanych parametrach (bez gwarancji, że rozwiązanie istnieje). Ręcznie zadano stacje końcowe tak, żeby rozwiązanie było znajdowane pod koniec przeszukiwania.

- `pod0.in` — przykład z treści zadania;
- `pod1.in` — $n=10$, stacja początkowa jest też stacją końcową;
- `pod2.in` — $n=10$, $k=5$, $q=20$, losowy (k jest liczbą linii komunikacyjnych);
- `pod3.in` — $n=20$, $k=3$, $q=30$, losowy;
- `pod4.in` — $n=100$, $k=1$, $q=90$, podróż trwa dokładnie 24h;
- `pod5.in` — $n=6$, $k=30$, $q=90$, losowy;
- `pod6.in` — $n=200$, $k=300$, $q=1000$, losowy;
- `pod7.in` — $n=300$, $k=2000$, $q=4000$, losowy;
- `pod8.in` — $n=500$, $k=100$, $q=4000$, losowy;
- `pod9.in` — $n=1000$, $k=200$, $q=2000$, losowy;
- `pod10.in` — $n=1000$, $k=500$, $q=4000$, losowy.