Zawody II stopnia

opracowania zadaĆń

Piotr Chrząstowski-Wachtel
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Sieć dróg

Do mapy drogowej dołączona została dyskietka z tabelą długości najkrótszych dróg (odległości) między każdymi dwoma miastami na mapie. Wszystkie drogi są dwukierunkowe. Położenie miast na mapie ma następującą ciekawą własność. Jeżeli długość najkrótszej drogi z miasta A do B jest równa sumie długości najkrótszych dróg z A do C i z C do B, to miasto C leży na (pewnej) najkrótszej drodze z A do B. Powiemy, że dwa miasta A i B sąsiadują ze sobą, jeżeli nie istnieje miasto C takie, że długość najkrótszej drogi z miasta A do B jest równa długości najkrótszych dróg z A do C i z C do B.
Na podstawie danej tabeli odległości znajdź wszystkie pary miast sąsiadujących ze sobą.

Przykład

Jeżeli tabela odległości ma postać

	A	B	C
A	0	1	2
B	1	0	3
C	2	3	0

wówczas sąsiednimi miastami są A i B oraz A i C.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego SIE.IN tabelę odległości;*
- *znajduje wszystkie pary sąsiednich miast;*
- *zapisuje wynik w pliku tekstowym SIE.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego SIE.IN znajduje się liczba całkowita n, $1 \leqslant n \leqslant 200$. Jest to liczba miast na mapie. Miasta ponumerowane są jednoznacznie od 1 do n.
Tabela odległości jest zapisana w kolejnych n wierszach pliku SIE.IN. W wierszu $i+1$, $1 \leqslant i \leqslant n$, pliku SIE.IN jest zapisanych n nieujemnych liczb całkowitych nie większych od 200, oddzielonych pojedynczymi odstępami. Liczba j-ta jest odległością między miastami i oraz j.

Wyjście

Twój program powinien wypisać w pliku wyjściowym SIE.OUT wszystkie pary (numerów) sąsiednich miast, po jednej parze w każdym wierszu.
Każda para powinna pojawić się tylko raz. Liczby w parze powinny być uporządkowane rosnąco i oddzielone pojedynczym odstępem.
Kolejność wypisywania par powinna być taka, że dla pary (a,b) poprzedzającej parę (c,d), $a < c$ lub $(a = c$ i $b < d)$.

Przykład

Dla pliku tekstowego SIE.IN:

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

poprawnym rozwiązaniem jest plik wyjściowy SIE.OUT:

```
1 2
1 3
```

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

Odległość między dwoma różnymi miastami jest zawsze większa od zera.

ROZWIĄZANIE

Zadanie powstało, gdy planowałem przejazd na wakacje samochodem nad morze. Wpatrywałem się w mapę drogową Polski i szukałem najkrótszej drogi z Warszawy do Świnoujścia. Zacząłem od wydrukowanej na odwrocie mapy tabelki odległości drogowych między miastami. Odczytałem tam, że do Świnoujścia mam 567 km, ale nijak nie mogłem wykombinować, jak tę wartość osiągnąć. Usiłowałem odszukać w tabelce takie miejscowości, przez które przejeżdżając będę miał w sumie te 567 km. Dobrze wyglądała międzynarodowa droga przez Poznań. Ale okazało się, że nie była ona najkrótsza. Przekonałem się o tym sprawdzając sumę odległości między Warszawą a Poznaniem i między Poznaniem a Świnoujściem. Z Warszawy do Poznania jest 308 km, a z Poznania do Świnoujścia 297 km — w sumie o 38 km za dużo. Podobnie było z Płockiem, Włocławkiem, Bydgoszczą, Piłą, Koszalinem — większymi miejscowościami, które optycznie leżą mniej więcej po drodze. Żadna z nich jakoś nie chciała mieć tej własności, żeby łączna liczba kilometrów z Warszawy do niej i od niej do Świnoujścia wyniosła 567.

W końcu znalazłem takie miasto: Toruń. Okazało się, że z Warszawy do Torunia jest 209 km, a z Torunia do Świnoujścia 358 km. Wiedziałem już, że powinienem jechać przez Toruń. Żeby opracować resztę trasy trzeba było rozwiązać to samo zadanie dla przejazdu z Warszawy do Torunia i z Torunia do Świnoujścia. Niestety, w zestawie większych miast, dla których była skonstruowana tabelka, nie było ani jednego między Warszawą a Toruniem i ani jednego między Toruniem a Świnoujściem, przez które prowadziłaby najkrótsza droga.

Oderwałem się więc od tabelki i wziąłem się za studiowanie mapy. Tabelka bowiem była zbyt zgrubna, żeby do końca podpowiedzieć marszrutę. Zawierała za mało miast. Pomyślałem sobie, że szkoda, bo gdyby zawierała wszystkie miasta w Polsce (a jest ich kilkanaście tysięcy), to wyszłoby ciekawe zadanie. W odróżnieniu od mapy, której reprezentacja graficzna jest niewygodna do automatyzacji, tabelka zawiera konkretne odległości, a dodatkowo posiada wszystkie dane, aby odtworzyć strukturę sąsiedztwa. Oczywiście sporządzenie takiej tabelki, w której uwzględnione będą wszystkie widniejące na mapie miasta w Polsce jest mało realne: przy n miastach rozmiar tabelki byłby wtedy równy $n(n+1)/2$ — mniej więcej połowa kwadratu ich liczby. Dałoby to wielkość rzędu setek milionów danych — za dużo, żeby w praktyce taka reprezentacja mogła być użyteczna. Dlatego na mapach podaje się zaledwie fragment takiej tabeli, wybierając jedynie niektóre ważniejsze węzły drogowe.

Znacznie prościej jest mapę drogową pamiętać podając odległości między sąsiednimi węzłami sieci samochodowej. Ze względu na to, że z takiego węzła możemy wyjechać przeważnie w 2–5 kierunkach, liczba zapisów dotyczących odległości łączących sąsiadów jest równa mniej więcej $5n$, czyli dla Polski kilkadziesiąt tysięcy. Oczywiście odtworzenie najkrótszych tras na podstawie takiej informacji byłoby zadaniem niejako dualnym do zadania konkursowego. Można się spodziewać, że część algorytmów rozwiązujących któryś z tych problemów będzie miała swoją dualną wersję rozwiązującą problem przeciwny. Dualne do naszego zadanie odtworzenia najkrótszych połączeń na podstawie opisu odległości między sąsiadami znane jest pod nazwą problemu szukania najkrótszych ścieżek w grafie. Tutaj mamy do czynienia ze specyficznym rodzajem grafu: takim, że wagi krawędzi łączących węzły są interpretowane, jako odległości. Na dobrą sprawę znajdowanie najkrótszej marszruty na podstawie tabelki odległości można rozbić na dwa etapy: najpierw odtworzyć strukturę sąsiedztwa, a następnie rozwiązać problem szukania najkrótszej ścieżki w grafie. Treścią zadania konkursowego był pierwszy etap.

Zadanie odtworzenia relacji sąsiedztwa jest proste. Wystarczy dla każdej pary węzłów sprawdzić warunek definiujący sąsiedztwo między dwoma węzłami. Dwa węzły są sąsiadami wtedy i tylko wtedy, gdy nie istnieje pomiędzy nimi węzeł odległy od nich w sumie o tyle, ile wynosi bezpośrednia odległość między nimi. Zorganizujmy więc trzy zagnieżdżone pętle: dwie zewnętrzne przebiegające zbiór par węzłów i najbardziej zagnieżdżoną przebiegającą wszystkie węzły i badającą, czy nie występują one na ścieżce między tamtą parą:

```
{tablica A zawiera odległości między miastami,
przy czym na przekątnej A[i,i] wstawione są jedynki}
for i:=1 to n − 1 do
  for j:=i + 1 to n do begin
    {badamy parę węzłów (i,j); ze względu na symetrię
    odległości możemy ograniczyć się do uporządkowanych par z i < j}
    sąsiedzi := true; {domniemamy, że i oraz j są sąsiadami}
    k := 1;
    while sąsiedzi and (k ⩽ n) do
      if A[i,k] + A[k,j] = A[i,j] then sąsiedzi := false;
      else k:=k + 1;
    if sąsiedzi then wypisz(i, j)
  end;
```

Jak widać, zadanie było wyjątkowo proste. Większość zawodników nie miała problemów z jego rozwiązaniem.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 12-tu testów SIE0.IN–SIE11.IN:

- SIE0.IN — test z treści zadania.
- SIE1.IN — najmniejszy możliwy zestaw danych.
- SIE2.IN, SIE7.IN — węzły ułożone na prostej; sąsiadami są węzły o kolejnych numerach.
- SIE3.IN, SIE8.IN — węzły ułożone gwiaździście wokół jednego wyróżnionego węzła; wyróżniony węzeł jest sąsiadem każdego innego. Żadne inne węzły nie sąsiadują ze sobą.
- SIE4.IN, SIE9.IN — węzły ułożone na pierścieniu; sąsiadami są węzły o kolejnych numerach oraz węzeł pierwszy z ostatnim.
- SIE5.IN, SIE10.IN — każde dwa węzły sąsiadują ze sobą.
- SIE6.IN, SIE11.IN — sąsiedztwa węzłów generowane losowo.

Testy były podzielone na trzy grupy, przy czym w ramach grup drugiej i trzeciej powtarzały się schematy generowania danych. Pierwszą grupę stanowiły testy podstawowe: test z treści zadania SIE0.IN i test SIE1.IN o rozmiarze najmniejszym z możliwych (dwa węzły połączone krawędzią).

Druga grupa (SIE2.IN–SIE6.IN) to testy o małych rozmiarach, dziesięciowęzłowe. Trzecia grupa (SIE7.IN–SIE11.IN) zawierała dokładnie takie same typy połączeń, jak grupa druga, tyle że dla 200 węzłów.

Wojciech Rytter
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Pakowanie kontenerów

Pewna fabryka pakuje swoje wyroby do pudełek, które mają kształt walca. Podstawy wszystkich pudełek są takie same. Wysokość każdego pudełka jest zawsze nieujemną, całkowitą potęgą dwójki, tzn. liczbą postaci 2^i, dla pewnego $i = 0, 1, 2, \ldots$. Liczbę i (wykładnik potęgi), która charakteryzuje wielkość pudełka, nazywamy rozmiarem tego pudełka. We wszystkich pudełkach znajduje się taki sam towar, ale w różnych pudełkach wartość towaru może być różna — towar wyprodukowany wcześniej jest tańszy. Dyrekcja fabryki wydała zarządzenie, żeby starać się wydawać najpierw towar najstarszy, czyli o najmniejszej wartości. Towar jest wywożony z magazynu w kontenerach. Każdy kontener, podobnie jak pudełka, ma kształt walca. Średnica kontenera jest niewiele większa od średnicy pudełek tak, że pudełka można swobodnie umieszczać w kontenerach. Wysokość każdego kontenera jest też nieujemną, całkowitą potęgą dwójki. Podobnie jak w przypadku pudełek, rozmiarem kontenera nazywamy wykładnik tej potęgi. Żeby bezpiecznie przewozić towar w kontenerze, należy go szczelnie wypełnić pudełkami, tzn. suma wysokości umieszczonych w kontenerze pudełek musi być równa wysokości tego kontenera.
Do magazynu dostarczono zestaw kontenerów. Sprawdź, czy można pudełkami z magazynu wypełnić szczelnie wszystkie kontenery. Jeśli tak, to podaj minimalną wartość towaru jaki można wywieźć z magazynu w tych, szczelnie wypełnionych, kontenerach.

Przykład

Rozważmy magazyn, w którym znajdują się 4 pudełka o następujących rozmiarach i wartościach przechowywanego w nich towaru:

```
1 3
1 2
3 5
2 1
1 4
```

Dwa kontenery o rozmiarach 1 i 2 można szczelnie wypełnić dwoma pudełkami o łącznej wartości 3, 4 lub 5, lub trzema pudełkami o łącznej wartości 9.
Kontener o rozmiarze 5 nie da się szczelnie wypełnić pudełkami z magazynu.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego PAK.IN charakterystyki pudełek w magazynie (rozmiar, wartość) oraz charakterystyki kontenerów (ile jest kontenerów danego rozmiaru);*
- *sprawdza, czy wszystkie kontenery w zestawie można szczelnie wypełnić pudełkami z magazynu, a jeśli tak, to oblicza minimalną wartość towaru jaką można w tych kontenerach wywieźć z magazynu;*
- *zapisuje wynik w pliku tekstowym PAK.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego PAK.IN znajduje się liczba całkowita n, $1 \leqslant n \leqslant 10000$. Jest to liczba pudełek w magazynie.
W każdym z n kolejnych wierszy zapisane są dwie nieujemne liczby całkowite oddzielone pojedynczym odstępem. Są to charakterystyki jednego pudełka, odpowiednio, rozmiar pudełka i wartość zawartego w nim towaru. Rozmiar pudełka jest nie większy od 1000, natomiast wartość jest nie większa od 10000.
W kolejnym wierszu podana jest dodatnia liczba całkowita q określająca, ile różnych rozmiarów mają kontenery dostarczone do magazynu.
W każdym z kolejnych q wierszy jest zapisana para dodatnich liczb całkowitych oddzielonych pojedynczym odstępem. Pierwsza z tych liczb jest rozmiarem kontenera, natomiast druga jest liczbą dostarczonych kontenerów o tym rozmiarze.

Wiadomo, że maksymalna liczba kontenerów nie przekracza 5000, a rozmiar kontenera jest nie większy od 1000.

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego PAK.OUT:

- *jedno słowo NIE, jeżeli nie da się szczelnie wypełnić pudełkami z magazynu kontenerów z podanego zestawu, lub*
- *jedną liczbę całkowitą, której wartość jest równa minimalnej, łącznej wartości towaru z pudełek, którymi można szczelnie wypełnić wszystkie kontenery z podanego zestawu.*

Przykład

Dla pliku wejściowego PAK.IN:

```
5
1 3
1 2
3 5
2 1
1 4
2
1 1
2 1
```

poprawnym rozwiązaniem jest plik wyjściowy PAK.OUT:

```
3
```

ROZWIĄZANIE

Rozwiązanie opiera się na pewnej własności sumowania całkowitych potęg dwójki. Niech liczba 2^k będzie sumą przynajmniej dwóch potęg dwójki:

$$2^k = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_s}, \qquad i_1 \leqslant i_2 \leqslant \ldots \leqslant i_s.$$

Wówczas wykładnik dowolnego składnika sumy jest mniejszy od k, a ponadto

$$i_1 = i_2.$$

Innymi słowy, składnik sumy o najmniejszym wykładniku musi się powtarzać.

Oznaczmy przez r minimalny rozmiar pudełka, a przez R minimalny rozmiar kontenera. Z opisanej własności sumowania potęg dwójki wynikają następujące własności optymalnego rozmieszczenia pudełek w kontenerach:

(1) $r = R$ — jeśli istnieje jakiekolwiek rozwiązanie, to istnieje rozwiązanie optymalne, w którym w najmniejszym kontenerze jest pudełko rozmiaru r o najmniejszej wartości,

(2) $r < R$ — jeśli istnieje jakiekolwiek rozwiązanie, to:

- jeśli jest tylko jedno pudełko rozmiaru r, to możemy je pominąć,
- jeśli są co najmniej dwa pudełka rozmiaru r, to możemy dwa z nich, najmniej wartościowe, zastąpić jednym o rozmiarze $r + 1$ i wartości będącej sumą wartości tych dwóch. Rozwiązanie optymalne dla tak zmodyfikowanego zestawu pudełek jest takie samo, jak dla zestawu wyjściowego.

(3) $r > R$ — zadanie nie ma rozwiązania, ponieważ najmniejszego kontenera nie da się zapełnić; pudełka są za duże w stosunku do niego.

Najciekawszy jest punkt (2) — mówi nam, że w pewnych sytuacjach dwa pudełka można **skleić** i traktować je zawsze jako jedno większe pudełko.

Z powyższych spostrzeżeń otrzymujemy **algorytm zachłanny** polegający na stosowaniu za każdym razem któregoś z punktów (1)–(3), w zależności od aktualnych wartości r i R. Albo zapełniamy całkowicie jakiś kontener (i liczba kontenerów maleje), albo sklejamy dwa pudełka (i liczba pudełek maleje), albo się zatrzymujemy, ponieważ zadanie nie ma rozwiązania.

Powyższa analiza pozwala już napisać program. W celu zmniejszenia złożoności zastosujemy pewną „sztuczkę": scalanie posortowanych list. Oznaczmy taką operację przez *scal*.

Załóżmy, że mamy listę *rozmiaryKontenerów*, na której zapamiętane są rozmiary kontenerów w kolejności niemalejącej (przy tym jeśli jest np. 6 kontenerów o rozmiarze 7, to na liście jest 6 siódemek pod rząd). Niech $pudełka_p, \ldots, pudełka_q$ są listami wartości pudełek o poszczególnych rozmiarach $p, \ldots, q$, gdzie p jest minimalnym rozmiarem pudełka, a q jest maksymalnym rozmiarem pudełka (na liście $pudełka_k$ przechowujemy pudełka o rozmiarze k posortowane względem wartości).

Załóżmy, że funkcja *pobierzPierwszy*(**var** *l* : *lista*) : *element* pobiera pierwszy (a więc najmniejszy) element z posortowanej listy (i zwraca jako wynik wywołania funkcji) i przy okazji usuwa ten element z listy. Umówmy się przy tym, że jeśli zastosujemy tę operację do pustej listy to algorytm się zatrzyma stwierdzając, że nie ma rozwiązania. Niech ponadto procedura *wstaw*(**var** *l* : *lista*; *e* : *element*) wstawia element na koniec listy.

Korzystamy z pomocniczej listy *podwójne*, na którą wrzucamy pudełka rozmiaru $r+1$ powstające przez sklejenie dwóch mniejszych pudełek z listy $pudełka_r$. Po wykorzystaniu listy $pudełka_r$ scalamy pomocniczą listę z listą $pudełka_{r+1}$.*

```
...
{przydziel pudełka do list pudełka_p, ..., pudełka_q wg ich rozmiarów}
{posortuj każdą z list pudełka_p, ..., pudełka_q wg wartości}
...
r := p;
podwójne := ∅;
wartość := 0;
brakRozwiązania := false;
while rozmiaryKontenerów ≠ ∅ and not brakRozwiązania do begin
  if pudełka_r = ∅ then begin
    scal(podwójne, pudełka_{r+1});
    podwójne := ∅
  end;
  R := min(rozmiaryKontenerów);
  r := min {i : pudełka_i ≠ ∅};
  if r = R then begin
    wartość := wartość + pobierzPierwszy(pudełka_r);
    _ := pobierzPierwszy(rozmiaryKontenerów);
  end
  else
    if r < R then begin
      if długość listy pudełka_r ⩾ 2 then
        wstaw(podwójne,
                    pobierzPierwszy(pudełka_r) + pobierzPierwszy(pudełka_r))
      else
        _ := pobierzPierwszy(pudełka_r)
    end
    else
      brakRozwiązania := true
end
if brakRozwiązania then wypisz('NIE')
```

* W opisie algorytmu, symbolem „_" (*podkreślenie*) oznaczono zmienną, której wartość nie jest w programie odczytywana (wykorzystaną w celu zapisania w języku Pascal wywołania funkcji z jednoczesnym zignorowaniem jej wyniku).

else *wypisz*(*wartość*)

Ciekawą własnością algorytmu jest to, że da się on zaimplementować w czasie liniowym. Gdybyśmy zawsze wstawiali **sklejoną** parę pudełek do listy pudełek większego rozmiaru, to za każdym razem potrzeba byłoby około $\log n$ kroków, w sumie $n \cdot \log n$. My robimy takie wstawienia „hurtowo", zamiast wstawiania pojedynczych elementów scalamy (za pomocą operacji *scal*) listy złożone z potencjalnie bardzo wielu elementów. Scalanie list uporządkowanych można wykonać w czasie liniowym. Przyjmijmy, że każdy element scalanych list jest obciążany „jednostką kosztu". Zauważmy, że dla każdego r, każde pudełko o rozmiarze r jest albo umieszczane w kontenerze, albo usuwane, albo sklejane z innym pudełkiem o tym samym rozmiarze i umieszczane na liście *podwójne*. Wynika stąd, że sumaryczny koszt scalań jest proporcjonalny do liczby pudełek pojawiających się w trakcie działania algorytmu (zliczamy zarówno pudełka dane na początku, jak i powstałe w wyniku scalań). Nietrudno policzyć, że takich pudełek jest co najwyżej dwa razy tyle, ile pudełek początkowych.

Pudełka pamiętamy w listach posortowanych rosnąco wg wartości, osobno dla każdego z rozmiarów. Na początku pudełka są nieuporządkowane, więc musimy je posortować. Można to zrobić w czasie liniowym korzystając z faktu, że wartości pudełek są liczbami całkowitymi z małego przedziału. Pozwala na zastosowanie sortowania kubełkowego* po rozmiarach oraz po wartościach elementów (czyli leksykograficznie, tzn. jeżeli rozmiary dwóch elementów są równe, to porównujemy ich wartości).

Z powyższych rozważań wynika, że złożoność algorytmu jest liniowa.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11-tu testów. Testy PAK3.IN–PAK7.IN zostały wygenerowane automatycznie w taki sposób, by w algorytmach działających na listach rozmiarów, operacja scalania działała zawsze na listach o podobnych długościach oraz rozkładach wartości. Dla danej liczby pudełek n, liczby kontenerów m oraz rozmiaru najmniejszego pudełka r w takim teście występuje $d = \log_2(n+1)$ różnych rozmiarów. Jego postać wygląda w przybliżeniu następująco (C oznacza maksymalną dopuszczalną wartość pudełka na wejściu):

rozmiar r	$\frac{n}{2}$ pudełek o wartościach z $\langle \frac{C}{2^d}, \frac{C}{2^{d-1}} \rangle$
rozmiar $r+1$	$\frac{n}{4}$ pudełek o wartościach z $\langle \frac{C}{2^{d-2}} - \frac{C}{2^{d-1}}, \frac{C}{2^{d-2}} \rangle$, $\frac{n}{4}$ kontenerów
rozmiar $r+2$	$\frac{n}{8}$ pudełek o wartościach z $\langle \frac{C}{2^{d-3}} - \frac{C}{2^{d-1}}, \frac{C}{2^{d-3}} \rangle$, $\frac{n}{8}$ kontenerów
...	
rozmiar $r+d-1$	$\frac{n}{2^{d-1}}$ pudełek o wartościach z $\langle C - \frac{C}{2^{d-1}}, C \rangle$, $\frac{n}{2^{d-1}}$ kontenerów

Dla takiego testu w algorytmach operujących na listach rozmiarów, z rozmiaru $r+i$ do rozmiaru $r+i+1$ przechodzi około $\frac{n}{2^{i+2}}$ pudełek o wartościach z przedziału (mniej więcej) $\langle \frac{C}{2^{d-i-1}} - \frac{C}{2^{d-1}}, \frac{C}{2^{d-i-1}} \rangle$.

- PAK0.IN — test z treści zadania.
- PAK1.IN — prosty test poprawnościowy.
- PAK2A.IN — prosty test z odpowiedzią „NIE".
- PAK2B.IN — prosty test poprawnościowy, zgrupowany z testem PAK2A.IN
- PAK3.IN–PAK6.IN — testy opisane powyżej.
- PAK7.IN–PAK10.IN — testy wydajnościowe o dużych rozmiarach.

* zob. [12] str. 215–217

Wojciech Rytter	Krzysztof Diks	Adam Borowski
autor zadania	opracowanie	program wzorcowy

Równanie na słowach

Słowem dwójkowym nazywamy każdy niepusty ciąg złożony z 0 lub 1. Równanie na słowach ma postać $x_1x_2\ldots x_l = y_1y_2\ldots y_r$, *gdzie* x_i *i* y_j *sa cyframi dwójkowymi (0 lub 1) lub zmiennymi, to jest małymi literami alfabetu angielskiego. Dla każdej zmiennej jest ustalona długość słów dwójkowych, które można podstawiać w jej miejsce. Długość tę nazywamy długością zmiennej. Rozwiązanie równania na słowach polega na przypisaniu każdej zmiennej słowa dwójkowego o odpowiadającej tej zmiennej długości, w taki sposób, by po zastąpieniu zmiennych w równaniu przez przypisane im słowa, obie strony równania (słowa dwójkowe) były takie same.*

Dla danego równania na słowach policz ile jest różnych rozwiązań tego równania.

Przykład

Niech $4, 2, 4, 4, 2$ *będą długościami zmiennych* a, b, c, d, e *w poniższym równaniu:*

$$1bad1 = acbe$$

To równanie ma 16 różnych rozwiązań.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego ROW.IN liczbę równań i ich opisy;*
- *znajduję liczbę rozwiązań każdego równania;*
- *zapisuje wyniki w pliku tekstowym ROW.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego ROW.IN znajduje się liczba całkowita x, $1 \leqslant x \leqslant 5$. *Jest to liczba równań. Następne wiersze zawierają opisy* x *równań. Na każdy opis składa się 6 wierszy. Między kolejnymi opisami nie ma pustych wierszy. Każde równanie jest opisane w następujący sposób:*
W pierwszym wierszu opisu znajduje się liczba całkowita k, $0 \leqslant k \leqslant 26$. *Jest to liczba różnych zmiennych w równaniu. Przyjmujemy, że zawsze zmiennymi jest* k *pierwszych małych liter alfabetu angielskiego. W drugim wierszu jest zapisany ciąg* k *liczb całkowitych dodatnich oddzielonych pojedynczymi odstępami. Te liczby to długości* k *kolejnych zmiennych* $a, b, \ldots$ *występujących w równaniu. Trzeci wiersz opisu zawiera tylko jedną dodatnią liczbę całkowitą* l. *Jest to długość lewej strony równania, tj. długość słowa utworzonego z cyfr 0 lub 1 i jednoliterowych zmiennych. W następnym wierszu jest zapisana lewa strona równania jako ciąg* l *cyfr lub zmiennych bez odstępów między nimi. Następne dwa wiersze zawierają opis prawej strony równania.*
Pierwszy z tych wierszy zawiera dodatnią liczbę r. *Jest to długość prawej strony równania. Drugi z wierszy zawiera prawą stronę równania zapisaną w taki sam sposób jak jego lewa strona. Liczba cyfr plus suma długości wszystkich zmiennych (licząc wszystkie wystąpienia każdej zmiennej) po każdej stronie równania nie przekracza 10000.*

Wyjście

Dla każdego i, $1 \leqslant i \leqslant x$, *Twój program powinien zapisać w* i*-tym wierszu pliku wyjściowego ROW.OUT liczbę różnych rozwiązań* i*-tego równania.*

Przykład

Dla pliku tekstowego ROW.IN:

```
1
5
```

```
4 2 4 4 2
5
1bad1
4
acbe
```

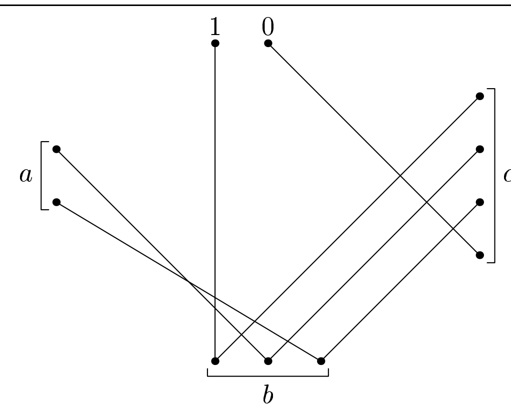
poprawnym rozwiązaniem jest plik wyjściowy ROW.OUT:

```
16
```

ROZWIĄZANIE

Rozwinięciem zmiennej x nazywamy ciąg $(x, 1), (x, 2), \ldots, (x, d_x)$, gdzie d_x jest długością tej zmiennej. **Rozwinięciem lewej (prawej) strony równania** nazywamy ciąg powstający w wyniku zastąpienia wystąpień wszystkich zmiennych po lewej (prawej) stronie równania przez ich rozwinięcia. Jeżeli długości rozwinięć obu stron są różne, równanie nie ma rozwiązania. Przyjmijmy zatem, że te długości są takie same. Problem obliczenia liczby rozwiązań danego równania sprowadza się do policzenia liczby spójnych składowych w **grafie zależności** tego równania. Wierzchołkami grafu zależności są pozycje w rozwinięciu zmiennych z równania i dwa symbole — cyfry „0" i „1". Dokładniej, jeśli x jest zmienną w równaniu, to w grafie zależności tego równania występują wierzchołki $(x, 1), (x, 2), \ldots, (x, d_x)$. Krawędzie w grafie zależności tworzymy w następujący sposób: dla każdej pozycji p w rozwinięciu lewej strony równania łączymy krawędzią element występujący w tym rozwinięciu na pozycji p z elementem występującym na tej samej pozycji w rozwinięciu prawej strony równania.

Rys. 1 Graf zależności dla równania $1ab0 = bc$ z przykładu.



Na rysunku 1 przedstawiono graf zależności dla równania

$$1ab0 = bc,$$

w którym długości zmiennych wynoszą:

$$a = 2, \qquad b = 3, \qquad c = 4.$$

Nietrudno zauważyć, że równanie nie ma rozwiązania wtedy i tylko wtedy, gdy w grafie zależności wierzchołki „0" i „1" należą do tej samej spójnej składowej. Jeżeli „0" i „1" należą do różnych spójnych składowych, to liczba rozwiązań równania jest równa dwa do potęgi będącą liczbą składowych w grafie zależności, które nie zawierają, ani „0", ani „1". Liczba rozwiązań równania z przykładu wynosi 4.

Z naszych rozważań wynika, że algorytm obliczania liczby rozwiązań danego równania na słowach jest bardzo prosty:

(1) Jeśli długości rozwinięć lewej i prawej strony równania są różne, to STOP — równanie nie ma rozwiązania.

(2) Zbuduj graf zależności równania.

(3) Jeśli do jednej składowej w grafie zależności należą „0” i „1”, to STOP — równanie nie ma rozwiązania.

(4) Liczba rozwiązań równania wynosi

$$2^{\text{liczba składowych w grafie zależności}}.$$

Spójne składowe grafu można obliczyć na jeden ze sposobów przedstawionych w opisie zadania Prostokąty. W rozwiązaniu wzorcowym do wyszukiwania spójnych składowych użyto przeszukiwania wszerz.

TESTY

Do sprawdzania rozwiązań zawodników użyto 12 testów ROW0.IN–ROW11.IN.

- ROW0.IN — test z treści zadania.
- ROW1.IN — dwa równania, jedno z nich nie zawiera żadnych zmiennych.
- ROW2.IN — dwa równania, jedno z nich nie ma rozwiązania.
- ROW3.IN — graf zależności składa się z trzech drzew i jednego cyklu.
- ROW4.IN — graf zależności składa się z dwóch składowych: cyklu 4-wierzchołkowego i dwóch cykli połączonych wspólnym wierzchołkiem.
- ROW5.IN — graf zależności posiada 10000 spójnych składowych; liczba rozwiązań znacznie (!!!) przekracza zasięg standardowych typów całkowitoliczbowych.
- ROW6.IN — w grafie zależności krawędzie można uzyskać na wiele różnych sposobów.
- ROW7.IN — dwa równania, dla jednego z nich graf zależności składa się z jednej spójnej składowej, w której wierzchołki “0” i “1” leżą daleko od siebie.
- ROW8.IN — graf zależności zawiera 125 składowych — 124 z nich to pojedyncze krawędzie, a 125-ta ma 3000 wierzchołków i około 6000 krawędzi.
- ROW9.IN, ROW10.IN — testy wydajnościowe o dużych rozmiarach.
- ROW11.IN — dwa równania, w pierwszym z nich długości rozwinięć lewej i prawej strony są różne.

Wojciech Guzicki
autor zadania, opracowanie

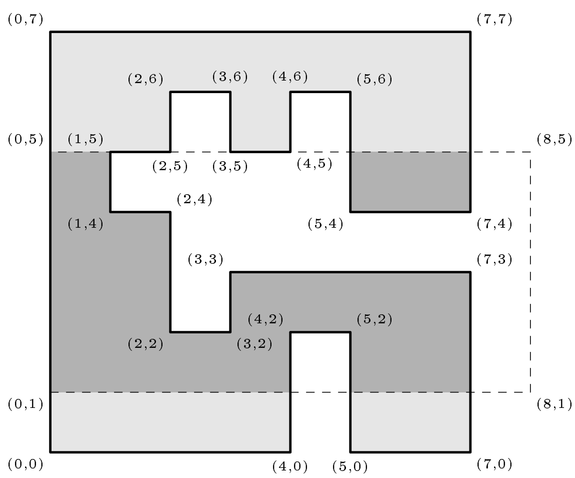
Grzegorz Jakacki
program wzorcowy

Okno

W prostokątnym układzie współrzędnych dany jest wielokąt o bokach równoległych do osi układu. Każde dwa kolejne boki w tym wielokącie są do siebie prostopadłe, a każdy wierzchołek ma współrzędne całkowitoliczbowe. Dane jest również okno, czyli prostokąt o bokach równoległych do osi układu. Wnętrze wielokąta (ale nie jego brzeg) zostało pokolorowane na czerwono. Jaka jest liczba rozłącznych czerwonych fragmentów wielokąta widzianych przez okno?

Przykład

Przyjrzyjmy się rysunkowi poniżej:



Liczba parami rozłącznych fragmentów wielokąta widzianych przez okno wynosi 2.

Zadanie

Napisz program, który: wczytuje opisy okna i wielokąta z pliku tekstowego OKN.IN; oblicza liczbę parami rozłącznych czerwonych fragmentów wielokąta widzianych przez okno; zapisuje wynik w pliku tekstowym OKN.OUT.

Wejście

W pierwszym wierszu pliku wejściowego OKN.IN znajdują się cztery liczby całkowite x_1, y_1, x_2, y_2 z przedziału [0..10000] oddzielone pojedynczymi odstępami.
Liczby x_1, y_1 są współrzędnymi górnego-lewego rogu okna. Liczby x_2, y_2 są współrzędnymi dolnego-prawego rogu okna.
Następny wiersz pliku wejściowego zawiera liczbę całkowitą n, $4 \leqslant n \leqslant 5000$. Jest to liczba wierzchołków wielokąta.
W kolejnych n wierszach znajdują się współrzędne kolejnych wierzchołków wielokąta danych w kierunku przeciwnym do ruchu wskazówek zegara, tzn. wnętrze wielokąta leży po lewej stronie jego brzegu, gdy przesuwamy się wzdłuż boków wielokąta zgodnie z zadanym porządkiem.
Każdy wiersz zawiera dwie liczby całkowite x, y oddzielone pojedynczym odstępem, $0 \leqslant x \leqslant 10000$, $0 \leqslant y \leqslant 10000$.
Liczby w wierszu $i+2$, $1 \leqslant i \leqslant n$, w pliku OKN.IN są współrzędnymi i-tego wierzchołka wielokąta.

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego OKN.OUT jedną liczbę całkowitą, a mianowicie liczbę rozłącznych czerwonych fragmentów wielokąta widzianych przez okno.

Przykład

Dla pliku wejściowego OKN.IN:

```
0 5 8 1
24
0 0
4 0
4 2
5 2
5 0
7 0
7 3
3 3
3 2
2 2
2 4
1 4
1 5
2 5
2 6
3 6
3 5
4 5
4 6
5 6
5 4
7 4
7 7
0 7
```

poprawnym rozwiązaniem jest plik wyjściowy OKN.OUT:

```
2
```

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

Brzeg wielokąta jest łamaną zwykłą zamkniętą, co oznacza, że każdy wierzchołek należy do dokładnie dwóch (kolejnych) boków, a każdy inny punkt łamanej należy do dokładnie jednego boku.

ROZWIĄZANIE

Zadanie *Okno* jest wzorowane na problemach często występujących w grafice komputerowej. Przypuśćmy, że w przestrzeni mamy daną pewną liczbę brył, których rzut musimy narysować. Istnieją stosunkowo proste algorytmy pozwalające obliczyć współrzędne rzutów końców wszystkich odcinków, które mamy narysować. Następnie rysujemy odcinki o końcach w tych wyznaczonych punktach. Możemy jednak zażądać, by nasze pole widzenia było w pewien sztuczny sposób ograniczone, na przykład, gdy rysujemy tylko to, co widać z okna. Musimy więc narysować tylko fragment tego, co dotychczas, mianowicie ten fragment, który mieści się w pewnym ograniczonym fragmencie płaszczyzny. Najczęściej tym ograniczonym fragmentem płaszczyzny jest prostokąt. Nasze zadanie ma podobny charakter: należy rozpoznać widoczne fragmenty większej całości. Oczywiście to zadanie nabrałoby więcej realizmu, gdyby odpowiedzią miał być ciąg list wierzchołków widocznych w oknie wielokątów. Utrudniłoby to jednak i tak dość zawiłą implementację rozwiązania, a samo rozwiązanie nie byłoby jednoznaczne. Dlatego ograniczono się tylko do żądania wskazania liczby widocznych wielokątów.

Algorytm rozwiązania jest bardzo prosty. Spróbujmy sobie wyobrazić „ręczne” rozwiązanie, nie zastanawiając się na razie nad sposobem implementacji. Przyjmijmy, że powierzchnia rysunku jest tak duża, że nie możemy ogarnąć jej wzrokiem. Na przykład, wielokąt i okno są narysowane na dużym boisku, na które nie możemy spojrzeć z góry. Będziemy zliczać wielokąty zawarte w oknie spacerując po obwodach wielokątów widocznych w oknie. Zaczniemy od znalezienia jakiegokolwiek wierzchołka naszego dużego wielokąta z treści zadania, leżącego na zewnątrz okna. Jeśli takiego wierzchołka nie ma, to cały wielokąt znajduje się wewnątrz okna i jest on widoczny w całości: odpowiedzią jest 1, bo widoczny jest jeden wielokąt. Rozpoznanie tego przypadku jest łatwe: każdy napotkany wierzchołek leży wewnątrz okna, a to można stwierdzić dość łatwo. Przypuśćmy więc, że znaleźliśmy wierzchołek leżący na zewnątrz okna. Rozpoczynamy wędrówkę w kierunku zgodnym z podanym w zadaniu (tzn. w kierunku przeciwnym do ruchu wskazówek zegara, czyli w taki sposób, by wnętrze wielokąta leżało po naszej lewej stronie). Wędrujemy dotąd, aż dojdziemy do krawędzi okna. Oczywiście może się zdarzyć, że nasza droga nigdy nie przetnie krawędzi okna. Jest to możliwe w jednym z trzech przypadków:

(1) cały wielokąt leży wewnątrz okna i odpowiedzią jest 1;

(2) całe okno leży wewnątrz wielokąta i odpowiedzią jest 1;

(3) wnętrze okna jest rozłączne z wnętrzem wielokąta i odpowiedzią jest 0.

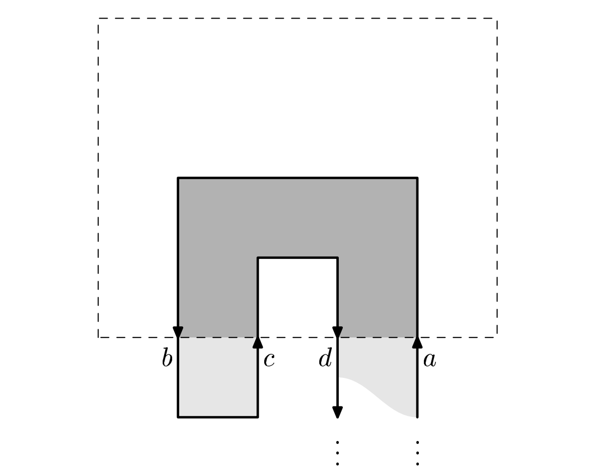
Kwestią, jak rozróżnić te przypadki, zajmiemy się później.

Tak więc kontynuujemy naszą wędrówkę dotąd, aż dojdziemy do granicy okna **i przekroczymy ją**. Zauważmy, że nie musimy przejmować się sytuacjami, w których dochodzimy do granicy okna, następnie skręcamy i podążamy wzdłuż tej granicy i jeszcze raz skręcamy w tę stronę, z której przyszliśmy (tzn. wracamy na zewnątrz okna, jeśli doszliśmy do granicy okna z zewnątrz lub wracamy do wewnątrz, jeśli przyszliśmy z wnętrza okna). Wynika to stąd, że mamy zliczać te wielokąty, których wnętrza są zawarte w oknie. Przekraczamy więc granicę okna i wchodzimy do jego wnętrza. Teraz podążamy wzdłuż obwodu wielokąta do momentu, w którym przekroczymy granicę okna próbując wyjść na zewnątrz. Taki moment oczywiście musi nastąpić: zaczęliśmy[[[zaczeliśmy?]]] wędrówkę na zewnątrz, jesteśmy wewnątrz, więc musimy kiedyś znaleźć się znów na zewnątrz. Teraz nie opuszczamy okna (a więc porzucamy naszą dotychczasową strategię polegającą na obchodzeniu obwodu całego wielokąta, natomiast chodzimy po obwodach wielokątów widocznych w oknie), ale skręcamy w lewo (by nadal mieć wielokąt po naszej lewej stronie) i podążamy wzdłuż granicy okna do **najbliższego** punktu, w którym znów droga wzdłuż obwodu wielokąta wchodzi do wielokąta z zewnątrz. W tym punkcie powtarzamy poprzednie postępowanie: idziemy wewnątrz okna do najbliższego punktu wyjścia z wielokąta, skręcamy w lewo i znów idziemy do **najbliższego** punktu wejścia itd. Oczywiście kiedyś to postępowanie się zakończy: dojdziemy do pierwszego punktu wejścia do okna. Wtedy nasza wędrówka wokół jednego widocznego fragmentu się kończy i (pamiętaną) liczbę widocznych fragmentów aktualizujemy (tzn. zwiększamy o 1). Przykładowe sytuacje są pokazane na rysunkach 1 i 2. W każdym przypadku wchodzimy do okna w punkcie a, podążamy wewnątrz okna do punktu b, skręcamy w lewo i podążamy wdłuż granicy okna do punktu c, znów wchodzimy do okna i wewnątrz niego dochodzimy do punktu d, ponownie skręcamy w lewo i powracamy do punktu a. Widzimy, że nie ma znaczenia, czy wewnątrz okna skręcimy tak, że wyjdziemy „na lewo", czy „na prawo" od punktu a.

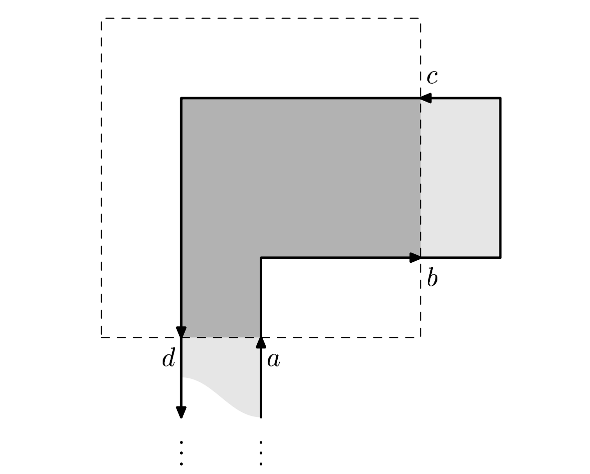
Teraz szukamy następnego punktu wejścia do okna i od niego zaczynamy tę samą procedurę. Robimy to dotąd, aż wyczerpiemy wszystkie punkty wejścia do okna i otrzymaną liczbę widocznych wielokątów podajemy jako wynik. Następnego wejścia do okna możemy szukać na różne sposoby. Jednym z nich jest obejście najpierw całego obwodu i zapamiętanie wszystkich punktów wejścia. W trakcie opisanej wyżej procedury usuwamy z listy punktów wejścia wszystkie te punkty, które wykorzystaliśmy. Po zamknięciu cyklu bierzemy pierwszy na liście punkt wejścia i od niego zaczynamy podobną wędrówkę.

Teraz już łatwo możemy się domyślić, w jaki sposób zaimplementować ten algorytm. Po pierwsze, odczytując z pliku wejściowego współrzędne wierzchołków wielokąta znajdujemy wszystkie punkty, w których przekraczamy granicę okna. Te punkty dzielimy na wejściowe (tzn. te, w których wchodzimy z zewnątrz do wnętrza wielokąta) i wyjściowe (gdy przekraczamy granicę okna w drugą stronę). Następnie tworzymy listę takich punktów wejścia i wyjścia. Dla punktu wejścia do okna następnym po nim na liście ma być odpowiedni punkt wyjścia (tzn. ten punkt wyjścia, który osiągniemy poruszając się wzdłuż obwodu wielokąta wewnątrz okna: dla punktów a i c na rysunkach 1 i 2 będą to odpowiednio punkty b i d). Dla punktu wyjścia następnym na liście ma być **najbliższy**, licząc w kierunku przeciwnym do ruchu wskazówek zegara, punkt wejścia do okna (na rysunkach 1 i 2 dla punktów b i d będą to odpowiednio punkty c i a). Widać więc, że wszystkie punkty wejścia musimy posortować (w programie wzorcowym

Rys. 1

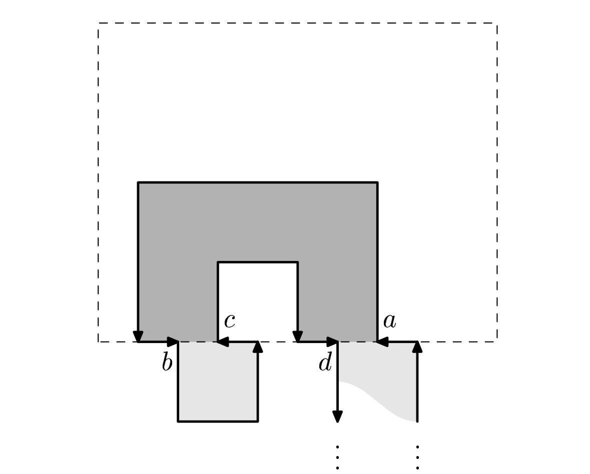


Rys. 2

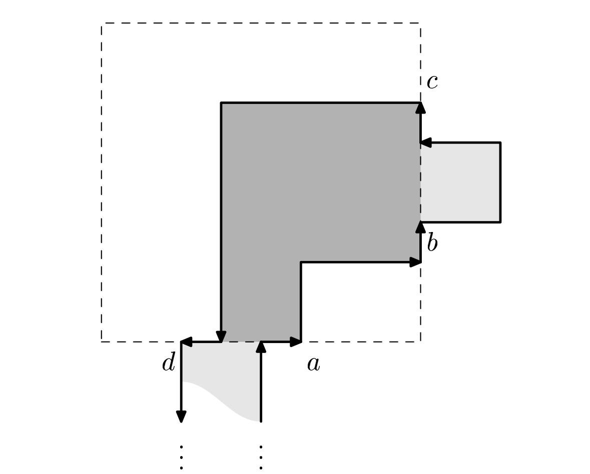


każdemu punktowi na obwodzie okna przypisujemy *wagę* w taki sposób, by wagi punktów zwiększały się przy obchodzeniu obwodu okna w kierunku przeciwnym do ruchu wskazówek zegara, a następnie sortujemy punkty względem tego atrybutu). Pewnej staranności wymaga wybór punktu przekroczenia granicy okna, gdy przez pewien czas poruszamy się wzdłuż tej granicy; porównaj rysunki 1 i 2 z odpowiednimi rysunkami 3 i 4. Należy zauważyć, że w punktach b i d już nie skręcamy w lewo — po prostu przechodzimy do następnego punktu wejścia, nie przejmując się nawet tym, że pewne odcinki przejdziemy dwukrotnie (np. z punktu d do a na rysunku 4). Ważne są tylko same punkty wejścia i wyjścia oraz ich uporządkowanie na obwodzie okna.

Rys. 3



Rys. 4

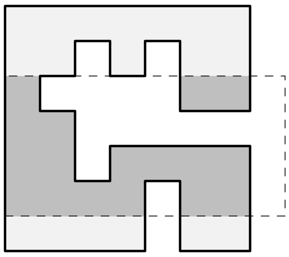


Ostatnim problemem implementacyjnym jest kwestia rozróżnienia sytuacji (1), (2) i (3) w przypadku, gdy obwód wielokąta nie przecina obwodu okna. Zauważmy, że jeżeli obwód wielokąta nie przecina obwodu okna, to stwierdzenie, czy wielokąt znajduje się w całości w oknie, sprowadza się do sprawdzenia, czy dowolny jego punkt (np. „pierwszy" punkt obwodu wielokąta) leży w oknie. To pozwala nam stwierdzić, czy mamy do czynienia z przypadkiem (1). Jeżeli jednak okaże się, że punkt obwodu wielokąta nie leży wewnątrz okna, to musimy jeszcze rozstrzygnąć pomiędzy przypadkiem (2) i (3). W tym celu wystarczy wybrać dowolny punkt okna (na przykład lewy dolny róg) i sprawdzić, czy znajduje się on wewnątrz wielokąta. Jak to zrobić? Bierzemy prostą pionową przechodzącą przez ten róg okna i sprawdzamy, czy poniżej rogu przecięła ona obwód wielokąta. Jeśli nie, to róg okna leży na zewnątrz wielokąta. Jeśli tak, to możemy postąpić dwojako: albo policzyć, ile razy ta prosta przecięła obwód wielokąta (róg okna leży wewnątrz wielokąta wtedy i tylko wtedy, gdy obwód wielokąta przetniemy nieparzystą liczbę razy), albo nieco prościej, wybrać najwyższy punkt przecięcia (tzn. leżący najbliżej rogu) i sprawdzić, w którym kierunku obwód wielokąta przebija tę prostą. Szczegóły tego rozwiązania pozostawiamy Czytelnikowi.

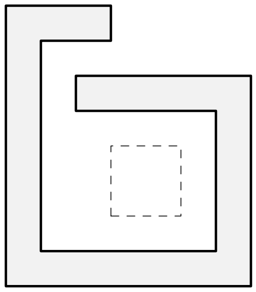
Warto jeszcze wspomnieć o wyborze metody sortowania. Może się zdarzyć, że obwód wielokąta przecina brzeg okna w bardzo wielu punktach (nawet powyżej tysiąca). Wtedy od wyboru metody sortowania zależy w głównej mierze szybkość działania programu. W programie wzorcowym wybrano metodę sortowania szybkiego (ang. *quicksort*).

TESTY

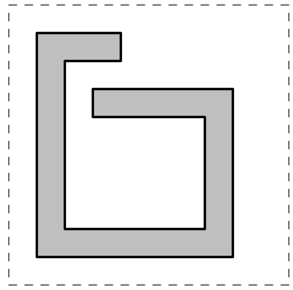
Rozwiązania zawodników oceniano przy pomocy zestawu 16-tu testów: PRO0.IN–PRO15.IN. Test PRO0.IN był testem z treści zadania. W teście PRO1.IN wielokąt był w całości poza oknem, w teście PRO2.IN w całości wewnątrz okna, w teście PRO3.IN pokrywał się z oknem. Testy PRO4.IN i PRO5.IN miały na celu sprawdzenie, czy program radzi sobie z sytuacjami, w których obwód wielokąta częściowo pokrywa się z brzegiem okna. Test PRO6.IN to długi meander przecięty oknem (21 widocznych kawałków). Test PRO7.IN miał sprawdzić poprawność programu na małym przykładzie, zaś testy PRO8.IN i PRO9.IN na przykładzie dwóch „spiral", mniejszej i większej. Testy PRO10.IN i PRO11.IN wykorzystywały „spirale" podwójnie skręcone, stosunkowo niewielkich rozmiarów (współrzędne wierzchołków nie przekraczały 240). Wielokąt w teście PRO12.IN miał ten sam kształt, co w teście PRO10.IN, ale był dziesięciokrotnie powiększony. Testy PRO13.IN, PRO14.IN i PRO15.IN miały maksymalne rozmiary danych i były to dwie spirale i jeden meander. Na kolejnych stronach przedstawiono graficzne schematy niektórych testów.



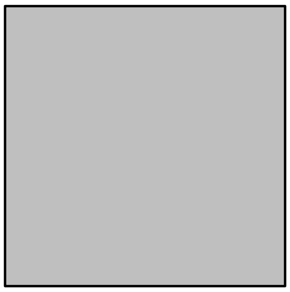
OKN0.IN



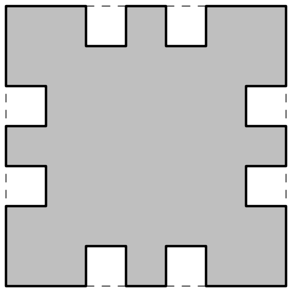
OKN1.IN



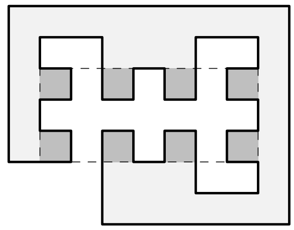
OKN2.IN



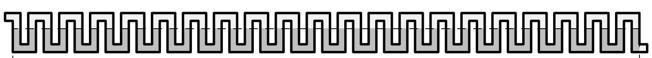
OKN3.IN



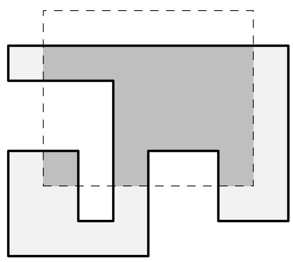
OKN4.IN



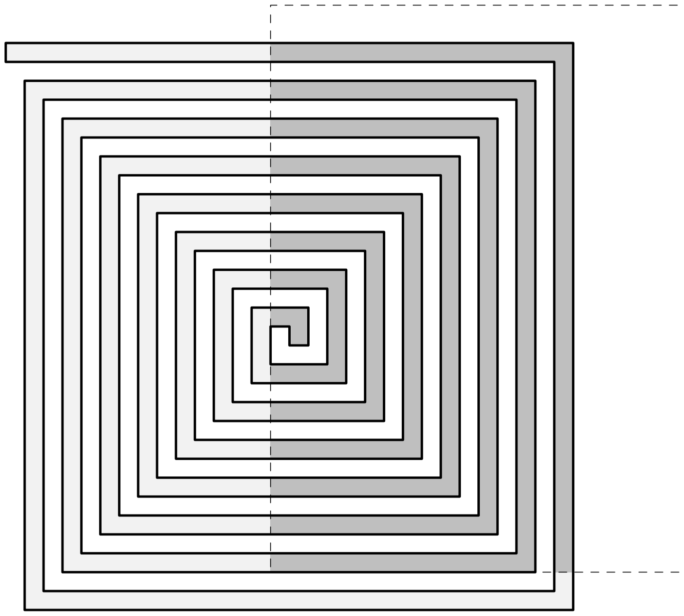
OKN5.IN



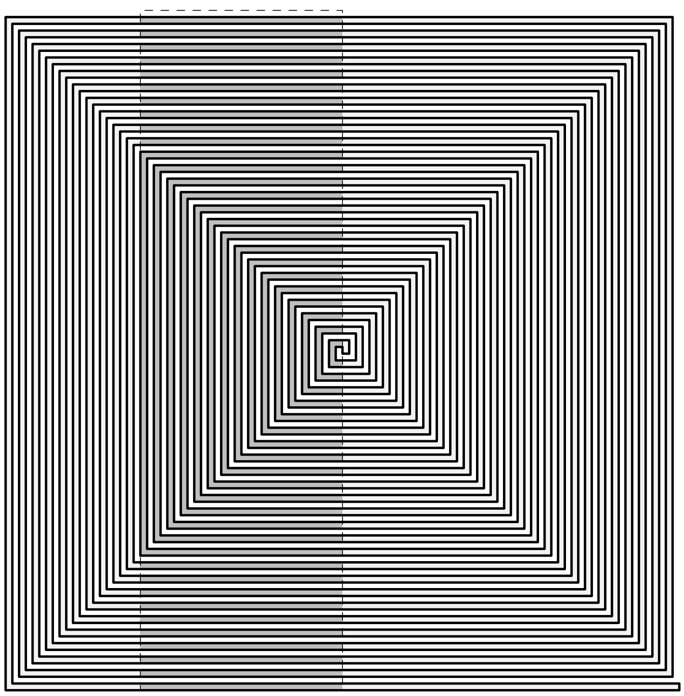
OKN6.IN



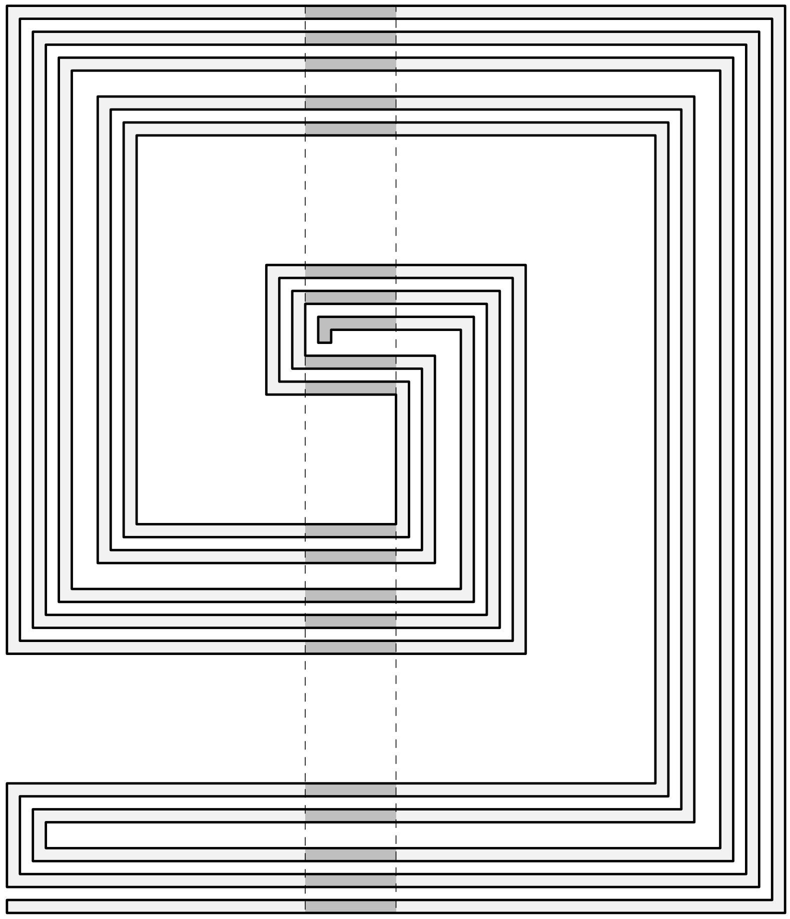
OKN7.IN



OKN8.IN



OKN9.IN



OKN10.IN

Autor nieznany

Tomasz Błaszczyk
program wzorcowy, opracowanie

Płetwonurek

Płetwonurek do nurkowania używa butli, w której są dwa zbiorniki: z tlenem i z azotem. W zależności od czasu przebywania pod wodą i głębokości nurek potrzebuje różnych ilości tlenu i azotu. Płetwonurek ma do dyspozycji pewną liczbę butli. Każda butla charakteryzuje się wagą oraz objętością zawartego w niej tlenu i azotu. Do wykonania zadania nurek potrzebuje określonych ilości tlenu i azotu. Jaka jest najmniejsza sumaryczna waga butli, które nurek musi zabrać ze sobą, żeby mógł wykonać zadanie?

Przykład

Nurek ma do dyspozycji 5 butli o następujących charakterystykach (odpowiednio: objętość tlenu w litrach, objętość azotu w litrach, waga butli w dekagramach):

```
3 36 120
10 25 129
5 50 250
1 45 130
4 20 119
```

Jeżeli do wykonania zadania nurek potrzebuje 5 litrów tlenu i 60 litrów azotu, to musi zabrać ze sobą dwie butle o łącznej wadze 249, np. pierwszą i drugą lub czwartą i piątą.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego PLE.IN zapotrzebowanie nurka na tlen i azot, liczbę dostępnych butli oraz ich charakterystyki;*
- *oblicza najmniejszą sumaryczną wagę butli, które nurek musi zabrać ze sobą, żeby wykonać zadanie;*
- *zapisuje wynik w pliku tekstowym PLE.OUT.*

Uwaga: dany zestaw butli zawsze gwarantuje wykonanie zadania.

Wejście

W pierwszym wierszu pliku wejściowego PLE.IN znajdują się dwie liczby całkowite t i a oddzielone pojedynczym odstępem, $1 \leqslant t \leqslant 21$ i $1 \leqslant a \leqslant 79$. Są to, odpowiednio, ilości tlenu i azotu potrzebne do wykonania zadania.
Drugi wiersz pliku wejściowego zawiera tylko jedną liczbę n, $1 \leqslant n \leqslant 1000$. Jest to liczba dostępnych butli.
Kolejne n wierszy zawiera charakterystyki butli.
Wiersz $i+2$ pliku PLE.IN zawiera trzy liczby całkowite t_i, a_i, w_i, pooddzielane pojedynczymi odstępami ($1 \leqslant t_i \leqslant t$, $1 \leqslant a_i \leqslant a$, $1 \leqslant w_i \leqslant 800$). Są to kolejno: objętości tlenu i azotu w i-tej butli oraz waga tej butli.

Wyjście

Twój program powinien zapisać jedną liczbę całkowitą w pierwszym i jedynym wierszu pliku wyjściowego PLE.OUT. Tą liczbą powinna być najmniejsza sumaryczna waga butli, które nurek musi zabrać ze sobą, żeby mógł wykonać zadanie.

Przykład

Dla pliku tekstowego PLE.IN:

```
5 60
5
3 36 120
```

```
10 25 129
5 50 250
1 45 130
4 20 119
```

poprawnym rozwiązaniem jest plik wyjściowy PLE.OUT:

```
249
```

ZMIANA W TREŚCI ZADANIA

Podczas trwania zawodów zostały zmienione ograniczenia: zamiast $1 \le t_i \le t$ i $1 \le a_i \le a$ wprowadzono $1 \le t_i \le 21$ i $1 \le a_i \le 79$.

ROZWIĄZANIE

Przy omawianiu rozwiązania zostaną użyte następujące oznaczenia:

- n — liczba butli,
- t, a — zapotrzebowanie na, odpowiednio, tlen i azot,
- $t[k]$, $a[k]$, $w[k]$ — liczba jednostek tlenu i azotu w k-tej butli oraz waga tej butli.

Zadanie można rozwiązać metodą programowania dynamicznego. Rozwiązanie zadania jest analogiczne do rozwiązania problemu plecakowego dla plecaka o ograniczonej pojemności (zob. [12], str. 381–383), rozszerzonego na przypadek dwuwymiarowy.

Algorytm działa w n fazach. W k-tej fazie rozwiązywane jest zadanie dla k pierwszych butli, na podstawie rozwiązania znalezionego w fazie poprzedniej, tj. dla $k-1$ pierwszych butli. Rozwiązania częściowe są przechowywane w tablicy T. Oto schemat algorytmu:

```
var
  T : array [0..t, 0..a] of word;
  ti, ai, td, ad, k : word;
begin
  T := [+∞, ..., +∞];
  T[0,0] := 0;
  for k := 1 to n do
    for ti := t downto 0 do
      for ai := a downto 0 do begin
        td := min(t, ti + t[k]);
        ad := min(a, ai + a[k]);
        T[td, ad] :=min(T[td, ad], T[ti, ai] + w[k])
      end;
  write(T[t, a])
end
```

Przez cały czas działania algorytmu spełniona jest następująca zależność (tzw. niezmiennik):

Po zakończeniu k-tej fazy każdy element $T[t', a']$ zawiera minimalną wagę zestawu butli wybranego spośród k pierwszych butli, w którym liczba jednostek tlenu wynosi:

$$\begin{cases} \textit{dokładnie } t' & \textit{gdy } t' < t \\ \textit{przynajmniej } t' & \textit{gdy } t' = t, \end{cases}$$

natomiast liczba jednostek azotu wynosi:

$$\begin{cases} \textit{dokładnie } a' & \textit{gdy } a' < a \\ \textit{przynajmniej } a' & \textit{gdy } a' = a. \end{cases}$$

Umawiamy się, że $+\infty$ oznacza, iż nie istnieje zestaw butli spełniający te warunki. Przyjmujemy także, że „po wykonaniu zerowej fazy" oznacza „przed wejściem do pętli".

Przed wykonaniem do wiersza nr 10 podtablica $T[0..(t_i + t[k] - 1),0..a]$ oraz fragment wiersza $T[t_i + t[k], 0..a_i + a[k]]$ zawierają rozwiązanie dla fazy $(k-1)$-wszej, natomiast pozostała część tablicy T — dla fazy k-tej.

W wierszach 10 i 11 obliczane są liczby jednostek tlenu i azotu, jakie można uzyskać dodając aktualnie rozpatrywaną butlę do zestawu umożliwiającego uzyskanie ti litrów tlenu i ai litrów azotu.

W wierszu 12 waga najlepszego znalezionego zestawu pozwalającego uzyskać td litrów tlenu i ad litrów azotu jest modyfikowana, jeżeli właśnie znaleziono zestaw o mniejszej wadze, niż najlżejszy znaleziony do tej pory. Stąd, po wykonaniu przypisania w wierszu 12, część tablicy zawierająca rozwiązanie dla fazy k-tej powiększa się o element $T[td, ad]$.

W algorytmie ważna jest kolejność przeglądania tablicy (wiersze 4 i 5). Wykazanie tego faktu pozostawiamy jako ćwiczenie dla Czytelnika.

Złożoność czasowa powyższego algorytmu wynosi $\Theta(tan)$, natomiast pamięciowa — $\Theta(ta)$.

Program wzorcowy jest bezpośrednią implementacją przedstawionego algorytmu. Zauważmy, że z podanych w zadaniu ograniczeń wynika, że waga zestawu butli może osiągnąć $79 \cdot 800 = 63200$ dekagramów, dlatego w Turbo Pascalu zamiast typu **integer** należy użyć typu **longint**.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 10-ciu testów.

- PLE0.IN — test z treści zadania.
- PLE1.IN — minimalne wymagania, jedna butla.
- PLE2.IN — test poprawnościowy, wartości zmiennych przekraczają zakres typu **integer**.
- PLE3.IN — mały test poprawnościowy.
- PLE4.IN — test faworyzujący rozwiązania heurystyczne, wystarcza jedna butla o wadze 101.
- PLE5.IN — test poprawnościowy; dwie grupy butli, opłaca się wybierać tylko z drugiej.
- PLE6.IN–PLE9.IN — różne testy wydajnościowe.