Zawody I stopnia

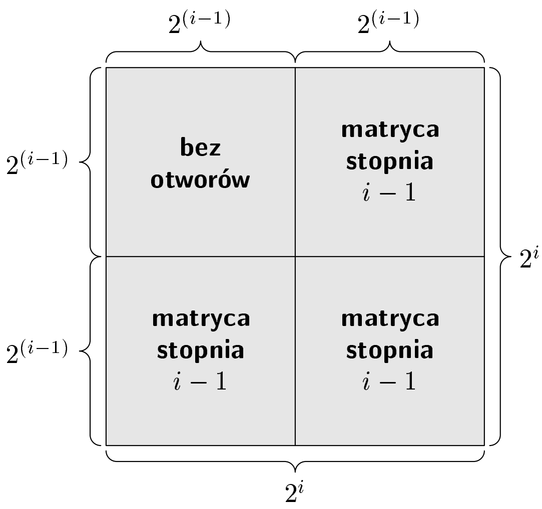
opracowania zadaĆń

Wojciech Rytter — autor zadania, opracowanie
Marcin Engel — program wzorcowy

Pracownia malarska

Pracownia malarska przygotowuje seryjną produkcję obrazów. Obrazy będą wykonywane za pomocą kwadratowych matryc o różnych stopniach. Matryca stopnia i składa się z 2^i wierszy i 2^i kolumn. Na przecięciu pewnych wierszy i kolumn znajdują się otwory. Matryca stopnia 0 ma jeden otwór.

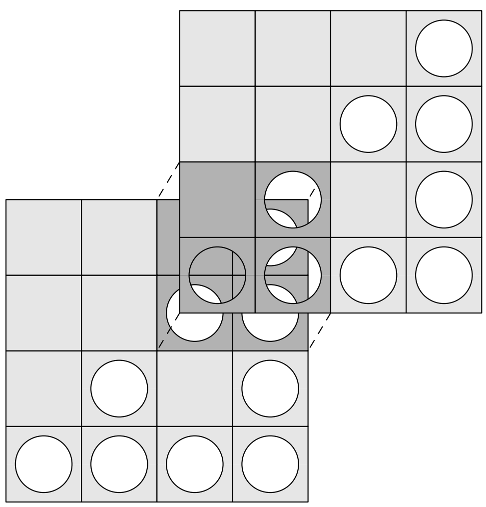
Dla $i > 0$ matryca stopnia i składa się z czterech kwadratów o rozmiarach $2^{(i-1)} \times 2^{i-1}$.



Oba prawe kwadraty oraz dolny lewy kwadrat są matrycami stopnia $i-1$. W górnym lewym kwadracie nie ma żadnych otworów. Obraz otrzymuje się w następujący sposób. Najpierw ustala się trzy nieujemne liczby całkowite n, x, y. Następnie umieszcza się dwie matryce stopnia n jedna na drugiej i górną matrycę przesuwa się o x kolumn w prawo i y wierszy w górę. Tak otrzymany wzorzec zostaje umieszczony na białym płótnie i na wspólną część obu matryc nanosi się żółtą farbę. W efekcie na płótnie pojawią się żółte plamy tylko w tych miejscach, w których otwory obu matryc pokrywają się.

Przykład

Przyjrzyj się dwóm matorycom stopnia 2 przedstawionym na rysunku.



Górna matryca została przesunięta o 2 kolumny w prawo i o 2 wiersze w górę. W trzech miejscach otwory z obu matryc pokrywają się.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego MAL.IN stopień obu matryc oraz współrzędne przesunięcia górnej matrycy;*
- *oblicza liczbę żółtych plam na płótnie;*
- *zapisuje wynik w pliku tekstowym MAL.OUT.*

Wejście

Pierwszy wiersz pliku tekstowego MAL.IN zawiera liczbę całkowitą n, $0 \leqslant n \leqslant 100$. Jest to stopień matryc używanych w produkcji obrazów.

W drugim wierszu zapisana jest liczba całkowita x, zaś w trzecim liczba całkowita y, $0 \leqslant x, y \leqslant 2^n$. Liczba x jest liczbą kolumn, a y jest liczbą wierszy, o które należy przesunąć górną matrycę.

Wyjście

W pierwszym wierszu pliku wyjściowego MAL.OUT należy zapisać liczbę plam na płótnie.

Przykład

Dla pliku wejściowego MAL.IN:

```
2
2
2
```

poprawnym rozwiązaniem jest plik wyjściowy MAL.OUT:

```
3
```

Twój program powienien szukać pliku MAL.IN w katalogu bieżącym i tworzyć plik MAL.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę MAL.???, grzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku MAL.EXE.

ROZWIĄZANIE

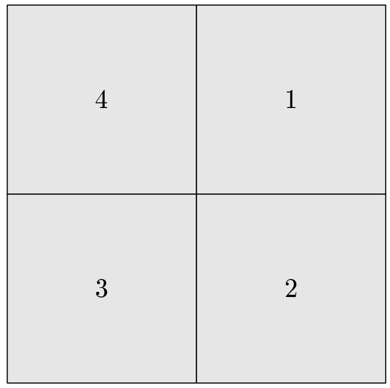
Oznaczmy przez M_i matrycę o rozmiarach $2^i \times 2^i$ skonstruowaną zgodnie z opisem zadania. Jej wiersze ponumerujmy z dołu do góry, a kolumny od lewej do prawej, rozpoczynając od zera. Ponumerujmy także ćwiartki matrycy, tak jak na rysunku 1.

Początkowa matryca ma rozmiary $2^n \times 2^n$. Wyróżnijmy pole matrycy o współrzędnych (x, y). Należy ono do jednej z czterech ćwiartek. Zapiszmy numer tej ćwiartki i odrzućmy resztę matrycy. Otrzymaliśmy znów matrycę, na której leży wyróżnione pole (być może ma teraz inne współrzędne). Powyższy proces możemy powtarzać, dopóki nie otrzymamy matrycy o rozmiarach 1×1. Zapisywane numery ćwiartek tworzą ciąg, który nazywać będziemy **adresem ćwiartkowym** pola (x, y).

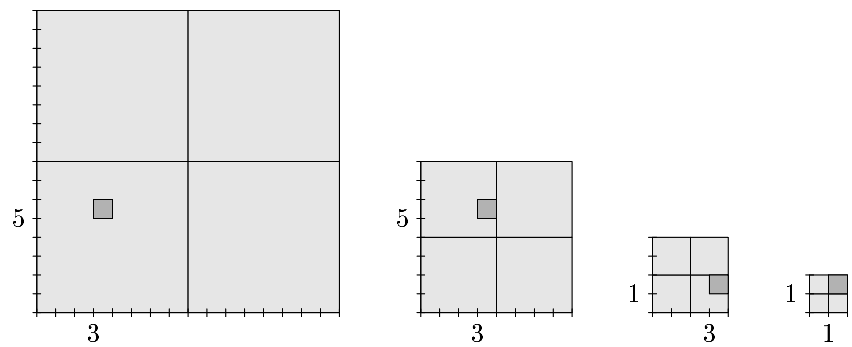
Skonstruujemy teraz algorytm, który będzie odpowiadał na pytanie nieco trudniejsze niż to postawione w treści zadania. Można w tym widzieć zjawisko podobne do występującego przy dowodzeniu przez indukcję, gdzie czasami łatwiej dowodzić silniejszej tezy indukcyjnej, gdyż tym samym możemy korzystać z silniejszego założenia indukcyjnego.

Spójrzmy na rysunek 3. Przedstawiono na nim cztery typy wspólnych części dwóch matryc stopnia i. Są to części wspólne matrycy stopnia i ze swoimi kopiami przesuniętymi o wektory $[x, y]$, $[x, y - 2^i]$, $[2^i - x, 2^i - y]$ i $[2^i - x, y]$. Będziemy je nazywać odpowiednio częściami wspólnymi typu I, II, III i IV.

Rys. 1 Numeracja ćwiartek matrycy.



Rys. 2 Sekwencja kolejnych położeń pola $(3,5)$ w coraz mniejszych ćwiartkach. Pole jest kolejno w ćwiartce trzeciej, czwartej, drugiej i pierwszej, stąd jego adresem ćwiartkowym jest $(3,4,2,1)$.



Niech $W_1(i)$, $W_2(i)$, $W_3(i)$ i $W_4(i)$ oznaczają odpowiednio liczby pokrywających się otworów w częściach wspólnych typu I, II, III i IV matryc stopnia i.

Załóżmy teraz, że punkt (x,y) leży w pierwszej ćwiartce matrycy stopnia i i spróbujmy wyrazić liczby $W_1(i)$, $W_2(i)$, $W_3(i)$, $W_4(i)$ przy pomocy liczb $W_1(i-1)$, $W_2(i-1)$, $W_3(i-1)$, $W_4(i-1)$. Wygodnie w tym celu posłużyć się pomocniczymi macierzami o rozmiarach 4×4. Wprowadźmy macierz

$$A_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

taką, że

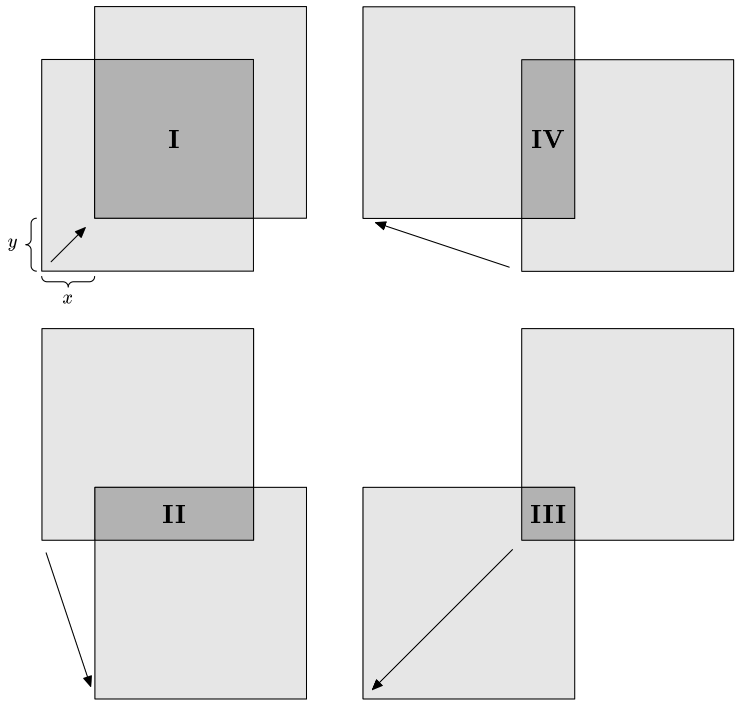
$$\begin{aligned} W_1(i) &= a_{11} \cdot W_1(i-1) + a_{12} \cdot W_2(i-1) + a_{13} \cdot W_3(i-1) + a_{14} \cdot W_4(i-1), \\ W_2(i) &= a_{21} \cdot W_1(i-1) + a_{22} \cdot W_2(i-1) + a_{23} \cdot W_3(i-1) + a_{24} \cdot W_4(i-1), \\ W_3(i) &= a_{31} \cdot W_1(i-1) + a_{32} \cdot W_2(i-1) + a_{33} \cdot W_3(i-1) + a_{34} \cdot W_4(i-1), \\ W_4(i) &= a_{41} \cdot W_1(i-1) + a_{42} \cdot W_2(i-1) + a_{43} \cdot W_3(i-1) + a_{44} \cdot W_4(i-1). \end{aligned}$$

Łatwo sprawdzić (umieszczając jedną matrycę nad drugą tak, by punkt przesunięcia był w pierwszej ćwiartce i badając położenia ćwiartek między sobą), że w tym przypadku

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

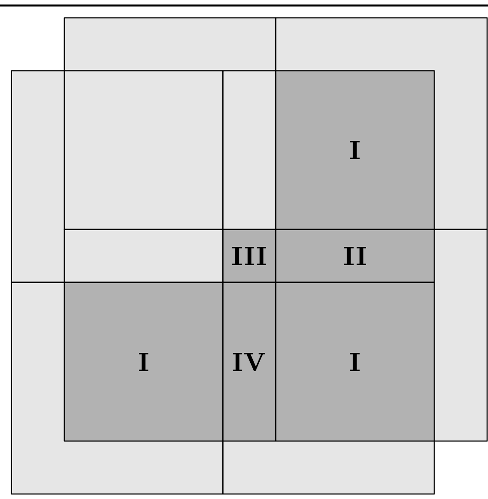
W k-tym wierszu macierzy trzeba zapisać, jak się wyraża $W_k(i)$ za pomocą $W_1(i-1)$, $W_2(i-1)$, $W_3(i-1)$ i $W_4(i-1)$. Pokażemy przykładowo, w jaki sposób obliczamy pierwszy wiersz macierzy A_3. W tym

Rys. 3 Wspólne części typu I, II, III i IV dwóch matryc o tym samym stopniu, przesuniętych względem siebie.



przypadku punkt (x, y) leży w trzeciej ćwiartce. Część wspólna składa się z trzech części typu I, oraz po jednej części typu II, III i IV (w sensie wspólnych części matryc stopnia $i-1$). Stąd pierwszym wierszem w macierzy A_3 jest $[3, 1, 1, 1]$ (zob. rysunek 4).

Rys. 4 Pierwszym wierszem macierzy A_3 jest $[3, 1, 1, 1]$, co oznacza, że jeżeli dolny lewy róg nałożonej matrycy znajduje się w pierwszej ćwiartce, to mamy trzy wspólne części typu I oraz po jednej wspólnej części typu II, III i IV dla matryc stopnia o jeden mniejszego. (Pamiętaj, że nie zaciemnione fragmenty części wspólnej nałożonych matryc nie zawierają otworów.)



W powyższy sposób wyznaczamy macierze A_1, A_2, A_3 i A_4 dla przypadków, gdy dolny lewy róg przesuniętej matrycy znajduje się odpowiednio w pierwszej, drugiej, trzeciej i czwartej ćwiartce. Są one podstawową strukturą danych potrzebną do rozwiązania zadania.

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 3 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 3 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad A_4 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Dla uproszczenia zapisu wprowadźmy jeszcze operację mnożenia wektora* przez macierz:

$$[v_1, v_2, v_3, v_4] \;\otimes\; \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = [r_1, r_2, r_3, r_4],$$

gdzie

$$\begin{aligned} r_1 &= [v_1 \cdot a_{11} + v_2 \cdot a_{21} + v_3 \cdot a_{31} + v_4 \cdot a_{41}], \\ r_2 &= [v_1 \cdot a_{12} + v_2 \cdot a_{22} + v_3 \cdot a_{32} + v_4 \cdot a_{42}], \\ r_3 &= [v_1 \cdot a_{13} + v_2 \cdot a_{23} + v_3 \cdot a_{33} + v_4 \cdot a_{43}], \\ r_4 &= [v_1 \cdot a_{14} + v_2 \cdot a_{24} + v_3 \cdot a_{34} + v_4 \cdot a_{44}]. \end{aligned}$$

Przy zastosowaniu powyższej notacji algorytm można zapisać następująco (zakładamy, że $[x, y]$ to wektor przesunięcia, a n to rozmiar matrycy):

```
1: adres := adresĆwiartkowy(x, y);
2: v := [1,0,0,0];
3: for t := 1 to n do begin
4:   i := adres[t];
5:   v := v ⊗ A_i
6: end;
7: wypisz(v[1])
```

Poprawność algorytmu można uzasadnić zauważając, że zawsze po wykonaniu linii 5 prawdziwy jest warunek (tzw. niezmiennik):

$$\mathit{wynik} = v[1] \cdot W_1(n-t) + v[2] \cdot W_2(n-t) + v[3] \cdot W_3(n-t) + v[4] \cdot W_4(n-t),$$

gdzie *wynik* oznacza wartość wypisywaną przez algorytm w linii 7. (Przyjmujemy $W_1(0) = 1$ i $W_2(0) = W_3(0) = W_4(0) = 0$.)

Mnożenie wektora przez macierz możemy zastąpić gotowym ciągiem operacji, na przykład mnożenie wektora v przez macierz A_4 sprowadza się do:

```
w[1] := v[1] + v[2];
w[2] := 3 · v[2];
w[3] := v[2] + v[3];
w[4] := v[1] + v[3];
v := w
```

Możemy wcześniej wyznaczyć ręcznie 4 ciągi operacji i zastąpić mnożenie przez macierz tymi operacjami. Będzie to na pewno bardziej efektywne obliczeniowo (w sensie czasu), chociaż notacja z macierzami wydaje się bardziej przekonywająca.

* Pisząc „wektor" mamy tu na myśli skończony, niekoniecznie dwulementowy, ciąg liczb. Tutaj nie interesuje nas interpretacja geometryczna wektora. Słowo „wektor" w takim znaczeniu jest powszechnie używane w algebrze liniowej.

Na przykład dla matrycy o rozmiarze 16×16 i przesunięcia [3,5] poprawną odpowiedzią będzie 14, co obliczamy następująco:

inicjacja:	$adres := [3,4,2,1]$
iteracja 1:	$[1,0,0,0] * A_3 := [3,1,1,1]$
iteracja 2:	$[3,1,1,1] * A_4 := [4,3,2,4]$
iteracja 3:	$[4,3,2,4] * A_2 := [8,6,6,12]$
iteracja 4:	$[8,6,6,2] * A_1 := [14,12,18,18]$
zakończenie:	$wypisz(14)$

TESTY

Oprócz testów poprawnościowych dla danych wejściowych o niedużych rozmiarach, przeprowadzono testy wydajnościowe dla danych o maksymalnym rozmiarze. Pozwoliło to na odsianie rozwiązań nadmiernie zużywających pamięć. Do sprawdzenia rozwiązań zawodników użyto 11-tu testów MAL0.IN–MAL10.IN.

- MAL0.IN — test z treści zadania.
- MAL1.IN–MAL3.IN, MAL10.IN — testy poprawnościowe dla $n \leqslant 5$ i x, y w przedziale $[0..31]$.
- MAL4.IN, MAL5.IN — testy dla przypadków brzegowych: przesunięcie zerowe, maksymalne przesunięcie po obu współrzędnych.
- MAL6.IN–MAL9.IN — testy wydajnościowe.

Krzysztof Diks
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Wielokąty

Powiemy, że dwa trójkąty przecinają się*, jeśli ich wnętrza mają co najmniej jeden punkt wspólny. Wielokąt jest* wypukły*, jeśli każdy odcinek łączący dowolne dwa punkty tego wielokąta jest w nim zawarty.*

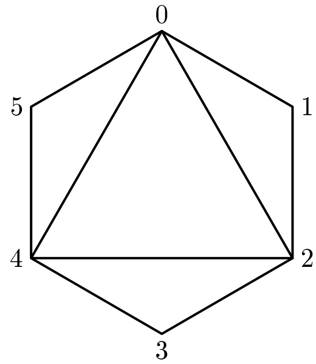
Trójkątem elementarnym *w wielokącie wypukłym nazywamy każdy trójkąt, którego wierzchołkami są wierzchołki tego wielokąta.*

Triangulacją *wielokąta wypukłego nazywamy każdy zbiór elementarnych trójkątów w tym wielokącie, w którym żadne dwa trójkąty nie przecinają się, a wszystkie razem pokrywają cały wielokąt.*

Dane są wielokąt wypukły i jego triangulacja. Jaka jest największa liczba trójkątów w tej triangulacji, które może przeciąć jeden elementarny trójkąt w tym wielokącie?

Przykład

Rozważmy następującą triangulację:



Trójkąt elementarny (1, 3, 5) przecina wszystkie trójkąty w tej triangulacji.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego WIE.IN liczbę wierzchołków wielokąta i jego triangulację;*
- *oblicza największą liczbę trójkątów tej triangulacji, które przecina pojedynczy, elementarny trójkąt w danym wielokącie;*
- *zapisuje wynik w pliku tekstowym WIE.OUT.*

Wejście

Pierwszy wiersz pliku WIE.IN zawiera liczbę n, $3 \leqslant n \leqslant 1000$. Jest to liczba wierzchołków wielokąta.

Wierzchołki wielokąta są ponumerowane kolejno $0, 1, \ldots, n-1$ zgodnie z ruchem wskazówek zegara.

Kolejne $n-2$ wiersze zawierają opisy trójkątów w triangulacji. W wierszu $i+1$, $1 \leqslant i \leqslant n-2$, zapisane są trzy liczby całkowite a, b, c oddzielone pojedynczymi odstępami. Są to numery wierzchołków i-tego trójkąta w triangulacji.

Wyjście

W pierwszym i jedynym wierszu pliku WIE.OUT należy zapisać jedną liczbę całkowitą — największą liczbę trójkątów w triangulacji, które przecinają jeden elementarny trójkąt w wielokącie wejściowym.

Przykład

Dla pliku wejściowego WIE.IN:

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

poprawnym rozwiązaniem jest plik tekstowy WIE.OUT:

```
4
```

Twój program powienien szukać pliku WIE.IN w katalogu bieżącym i tworzyć plik WIE.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę WIE.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku WIE.EXE.

ROZWIĄZANIE

Zadanie Wielokąt bardzo łatwo rozwiązać za pomocą algorytmu działającego w czasie rzędu n^4. W tym celu wystarczy dla każdego trójkąta elementarnego policzyć liczbę trójkątów w triangulacji, z którymi taki trójkąt przecina się, a następnie wybrać największą z obliczonych liczb. Ponieważ trójkątów w triangulacji n-kąta wypukłego jest zawsze $n-2$, a wszystkich trójkątów elementarnych jest $n(n-1)(n-2)/6$, to taki algorytm rzeczywiście działa w czasie rzędu n^4.

Istnieje znacznie lepszy algorytm dla naszego zadania. Algorytm, który opisujemy poniżej, działa w czasie liniowym ze względu na rozmiar danych wejściowych n.

Niech W będzie n-kątem wypukłym, a T jego dowolną triangulacją. Nietrudno zauważyć, że taka triangulacja jest jednoznacznie wyznaczona przez $n-3$ przekątnych wielokąta W, przecinających się co najwyżej w wierzchołkach wielokąta. Od tej chwili pisząc *przekątna* będziemy mieli zawsze na myśli przekątną ze zbioru przekątnych wyznaczających triangulację T.

Obserwacja 1: Niech K będzie dowolnym trójkątem elementarnym w wielokącie W i niech k będzie liczbą przekątnych, które mają niepuste przecięcie z wnętrzem K. Wówczas liczba tójkątów w triangulacji T, które przecinają K, wynosi $k+1$.

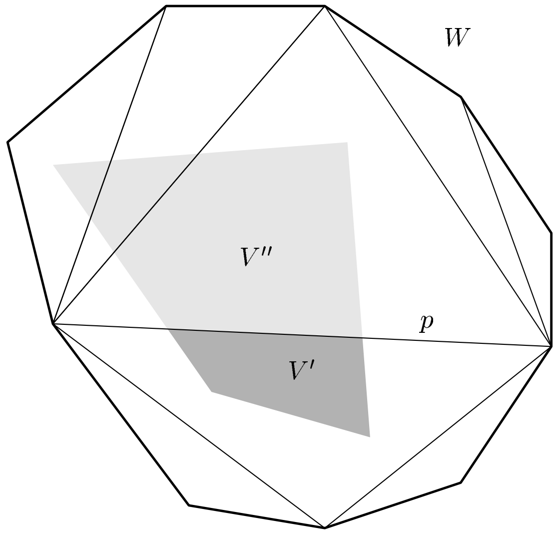
Obserwacja 1 wynika natychmiast z następującego ogólniejszego twierdzenia.

Twierdzenie 2: Niech V będzie dowolnym wielokątem wypukłym zawartym w W i niech v będzie liczbą przekątnych, które mają niepuste przecięcie z wnętrzem V. Wówczas, liczba trójkątów z T, które przecinają* V wynosi $v+1$ (zobacz rysunek 1).

Dowód: Indukcja po liczbie wierzchołków wielokąta W. Dla $n=3$ twierdzenie jest oczywiście prawdziwe. Niech $n>3$ i załóżmy, że twierdzenie jest prawdziwe dla każdego wielokąta W o mniejszej niż n liczbie wierzchołków. Jeśli wielokąta V nie przecina żadna przekątna, to jest on całkowicie zawarty w pewnym trójkącie z triangulacji T. Przyjmijmy zatem, że V przecina co najmniej jedna przekątna. Każda przekątna przecinająca wielokąt V dzieli go na dwa podwielokąty wypukłe (zobacz rysunek 1). Nietrudno zauważyć, że wśród przekątnych można znaleźć co najmniej jedną taką, która dzieli wielokąt na takie dwa podwielokąty, że jednego z nich nie przecina żadna przekątna. Niech p będzie taką przekątną, a V' i V'' będą podwielokątami V powstającymi w wyniku podziału V za pomocą przekątnej p. Załóżmy, że V' nie przecina żadna przekątna. Przekątna p dzieli też wielokąt W na dwa wielokąty wypukłe: W' — zawierający V', i W'' — zawierający V''. Niech T' będzie triangulacją T „obciętą" do wielokąta W', natomiast T'' triangulacją T „obciętą" do W''. Jeśli v'' jest liczbą przekątnych przecinających V'', to, z założenia indukcyjnego, liczba trójkątów z T'' (a stąd z T) przecinających V'' wynosi $v''+1$. Podwielokąt V' jest zawarty w dokładnie jednym trójkącie z T' (z T). Otrzymujemy zatem, że liczba trójkątów przecinających V wynosi $v''+2$. Z drugiej strony widzimy, że wielokąt V przecina $v=v''+1$ przekątnych, co dowodzi tezy twierdzenia.

* Przyjmujemy, że dwa wielokąty przecinają się, gdy ich wnętrza mają co najmniej jeden punkt wspólny.

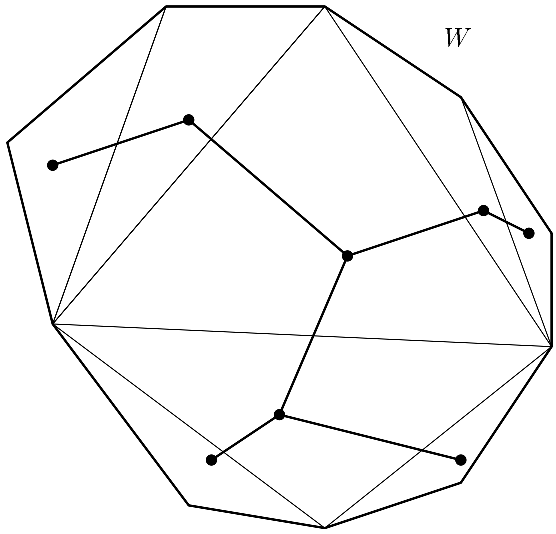
Rys. 1 Ilustracja dowodu twierdzenia 2



Zauważmy jeszcze, że liczba przekątnych o tej samej własności co p wynosi co najwyżej 3. □

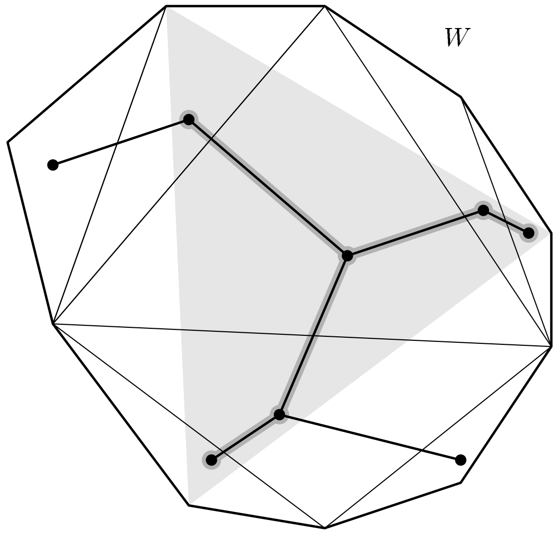
W jaki sposób wyznaczyć szybko trójkąt elementarny, który przecina największa liczba przekątnych z danej triangulacji? Żeby odpowiedzieć na to pytanie sformułujemy nasze zadanie w terminach teorii grafów.

Rys. 2 Drzewo triangulacji



Niech DT będzie grafem, którego wierzchołkami są trójkąty danej triangulacji T. Krawędź w grafie DT łączy dwa wierzchołki-trójkąty wtedy i tylko wtedy, gdy mają one wspólny bok (zobacz rysunek 2). Jest oczywiste, że z każdego wierzchołka wychodzą co najwyżej trzy krawędzie. Nietrudno zauważyć, że graf DT jest spójnym grafem bez cykli. Grafy o takich własnościach nazywa się **drzewami**. Graf DT będziemy nazywali **drzewem triangulacji**. Zauważmy, że krawędzie w drzewie DT odpowiadają przekątnym w triangulacji. Niech K będzie dowolnym trójkątem elementarnym i niech T_K będzie zbiorem wszystkich tych trójkątów z triangulacji T, które przecinają K. Oznaczmy przez $DT(T_K)$ taki maksymalny podgraf drzewa DT, którego wierzchołkami są tylko trójkąty z T_K. Można pokazać, że:

Rys. 3 Drzewo DT_K



Twierdzenie 3: Graf $DT(T_K)$ jest drzewem, w którym co najwyżej trzy wierzchołki mają stopień* 1. Pozostałe wierzchołki mają stopień 2, poza być może jednym, który ma stopień 3 (zobacz rysunek 3). □

Prawdziwe jest też twierdzenie "odwrotne".

Twierdzenie 4: Niech DT' będzie poddrzewem drzewa DT, w którym co najwyżej trzy wierzchołki maja stopień 1, a pozostałe są stopnia 2, poza być może jednym, który ma stopień 3. Wówczas istnieje trójkąt elementarny K taki, że $DT' = DT(T_K)$. □

Powyższe rozważania prowadzą nas do następującego algorytmu:

(1) Zbuduj drzewo triangulacji DT.

(2) Oblicz rozmiar (liczbę wierzchołków) największego poddrzewa drzewa DT, w którym co najwyżej 3 wierzchołki mają stopień 1, a pozostałe mają stopień 2, być może poza jednym, który ma stopień 3.

Zauważmy, że jeżeli w drzewie DT są co najmniej 3 wierzchołki stopnia 1, to największy rozmiar spośród poddrzew o podanych wyżej własnościach będzie miało poddrzewo, w którym są trzy wierzchołki stopnia 1 (a stąd wynika, że także dokładnie jeden wierzchołek stopnia 3), będące jednocześnie wierzchołkami o stopniu 1 w drzewie triangulacji DT.

Budowa drzewa triangulacji jest czysto techniczna i nie powinna nastręczać żadnych trudności. Format danych wejściowych umożliwia zbudowanie drzewa triangulacji w czasie liniowym. Szczegóły znajdzie Czytelnik w programie WIE.PAS na załączonej dyskietce. Tutaj omówimy ciekawszy problem efektywnej implementacji kroku (2). Rozważmy 3 przypadki:

- Drzewo DT składa się tylko z jednego wierzchołka. Ten przypadek zachodzi tylko dla $n = 3$ i odpowiedzią jest wówczas 1.
- Drzewo DT jest ścieżką, tzn. dokładnie dwa wierzchołki mają stopień 1, a pozostałe mają stopień 2. W tym przypadku poddrzewem o największym rozmiarze jest całe drzewo i poszukiwaną liczbą jest $n - 2$.
- W drzewie DT co najmniej 3 wierzchołki mają stopień 1. W takim drzewie musi istnieć wierzchołek o stopniu większym niż 1. Niech r będzie dowolnym takim wierzchołkiem. Na drzewo DT możemy teraz spojrzeć jak na drzewo z korzeniem r, w którym wszystkie krawędzie są skierowane w kierunku liści — wierzchołków o stopniu 1 w drzewie DT. Dla każdego wierzchołka v, niech DT_v będzie poddrzewem drzewa DT o korzeniu w v i zawierającym wszystkie wierzchołki, do których ścieżki z

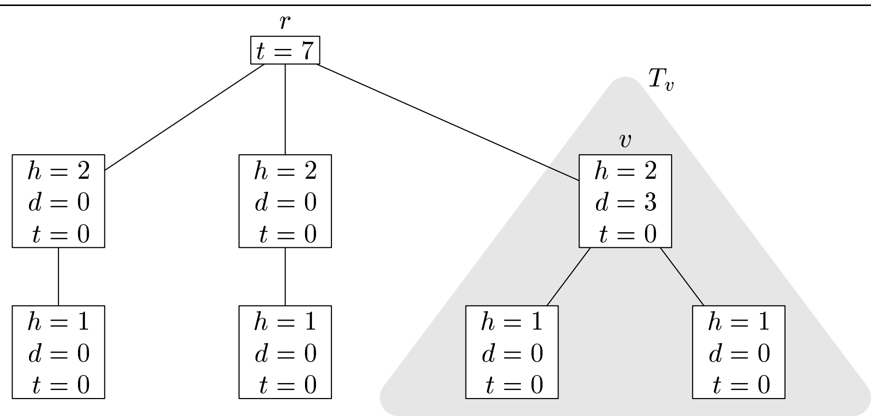
* Stopniem wierzchołka w grafie nazywamy liczbę wychodzących z niego krawędzi.

r prowadzą przez v (zobacz rysunek 4). W celu wyznaczenia rozmiaru największego poddrzewa w drzewie triangulacji, które zawiera dokładnie 3 wierzchołki o stopniu 1 i jeden wierzchołek o stopniu 3, a pozostałe wierzchołki mają stopień 2, obliczamy dla każdego wierzchołka v trzy wielkości:

- $h(v)$ — wysokość drzewa DT_v liczoną w liczbie wierzchołków;
- $d(v)$ — rozmiar największego poddrzewa w DT_v o korzeniu w v, zawierającego dokładnie dwa liście;
- $t(v)$ — rozmiar największego poddrzewa w DT_v, niekoniecznie o korzeniu w v, zawierającego dokładnie trzy liście.

Poszukiwaną liczbą jest $t(r)$.

Rys. 4 Ukorzenione drzewo triangulacji



Liczby $h(v)$, $d(v)$, $t(v)$ obliczamy rekurencyjnie korzystając z następujących zależności:

- jeśli v jest liściem, to:
$$h(v) = 1, \qquad d(v) = 0, \qquad t(v) = 0.$$
- jeśli v ma jedynego syna x, to:
$$h(v) = h(x) + 1, \qquad d(v) = d(x) + 1, \qquad t(v) = t(x).$$
- jeśli v ma dokładnie dwóch synów x i y, to:
$$\begin{aligned} h(v) &= \max(h(x),\ h(y)) + 1,\\ d(v) &= h(x) + h(y) + 1,\\ t(v) &= \max(t(x),\ t(y),\ h(x) + d(y) + 1,\ d(x) + h(y) + 1). \end{aligned}$$
- jeśli v ma dokładnie trzech synów x, y i z (przypadek ten zachodzi tylko wtedy, gdy v jest korzeniem drzewa ($v = r$), wystarczy więc policzyć tylko $t(v)$), to:
$$t(v) = \max \begin{cases} t(x)\\ t(y)\\ t(z)\\ d(x) + \max(h(y),\ h(z)) + 1\\ d(y) + \max(h(x),\ h(z)) + 1\\ d(z) + \max(h(x),\ h(y)) + 1\\ h(x) + h(y) + h(z) + 1 \end{cases}$$

UWAGA: dla uproszczenia zapisu przyjmujemy, że w przypadku gdy $d(\cdot)$ jest składnikiem sumy i ma wartość 0, to cała suma też ma wartość 0.

Ponieważ obliczenia w każdym wierzchołku zajmują stały czas, łączny czas obliczeń jest rzędu rozmiaru drzewa, czyli $O(n)$.

TESTY

Rozwiązania zawodników były sprawdzane za pomocą 11 testów WIE0.IN–WIE10.IN:

- WIE0.IN — test z treści zadania.
- WIE1.IN–WIE3.IN — przypadki szczególne: trójkąt, czworokąt, pięciokąt. Wynikiem jest zawsze $n-2$.
- WIE4.IN, WIE5.IN — testy poprawnościowe: drzewo triangulacji jest ścieżką.
- WIE6.IN, WIE7.IN — testy poprawnościowe: ukorzenione drzewa triangulacji przypominają pełne drzewa binarne o wysokościach, odpowiednio, 2 i 7.
- WIE8.IN–WIE10.IN — testy wydajnościowe: losowe triangulacje wielokątów o dużych rozmiarach.

Grzegorz Jakacki
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Suma ciągu jedynkowego

Ciąg liczbowy o wartościach będących liczbami całkowitymi nazywamy jedynkowym, *jeżeli dowolne jego sąsiednie wyrazy różnią się od siebie dokładnie o jeden oraz jego pierwszy wyraz jest równy 0. Bardziej precyzyjnie: niech* $[a_1, a_2, \ldots, a_n]$ *będzie ciągiem o wartościach całkowitych; powiemy, że ten ciąg jest* jedynkowy, *jeżeli:*

- *dla dowolnej liczby całkowitej* k *spełniającej nierówność* $1 \leqslant k < n$ *zachodzi warunek* $|a_k - a_{k+1}| = 1$ *oraz*
- $a_1 = 0$.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego SUM.IN dwie liczby całkowite: długość ciągu i sumę elementów ciągu;*
- *wyznaczy ciąg jedynkowy o zadanej długości i sumie elementów lub stwierdzi, że taki ciąg nie istnieje;*
- *zapsize rezultat w pliku tekstowym SUM.OUT.*

Wejście

W pliku tekstowym SUM.IN są zapisane:

- *w pierwszym wierszu — liczba* n *elementów ciągu, spełniająca nierówność* $1 \leqslant n \leqslant 10000$;
- *w drugim wierszu — liczba* S *będąca żądaną sumą elementów ciągu, spełniająca nierówność* $|S| \leqslant 50\ 000\ 000$.

Wyjście

W pierwszych n *wierszach pliku tekstowego SUM.OUT należy zapisać* n *liczb całkowitych (po jednej w wierszu) będących kolejnymi wyrazami ciągu jedynkowego (k-ty wyraz w k-tym wierszu) o zadanej sumie* S *lub słowo NIE, jeżeli taki ciąg nie istnieje.*

Przykład

Dla pliku wejściowego SUM.IN o zawartości:

```
8
4
```

poprawnym rozwiązaniem jest plik wyjściowy SUM.OUT o zawartości:

```
0
1
2
1
0
-1
0
1
```

Twój program powienien szukać pliku SUM.IN w katalogu bieżącym i tworzyć plik SUM.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę SUM.???, grzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku SUM.EXE.

ROZWIĄZANIE

Niech A będzie pewnym skończonym podzbiorem zbioru liczb naturalnych. Oznaczmy przez ΣA sumę* elementów zbioru A, a przez $\Sigma^* A$ zbiór sum elementów wszystkich podzbiorów A, tj. $\Sigma^* A = \{\Sigma X \;:\; X \subseteq A\}$. Jeżeli $A = \{2, 5, 7, 8\}$, to $\Sigma A = 22$ oraz $\Sigma^* A = \{0, 2, 5, 7, 8, 9, 10, 12, 13, 14, 15, 17, 20, 22\}$. Niech $\mathcal{S}_n$ będzie sumą wszystkich liczb naturalnych od 1 do n, tj. $\mathcal{S}_n = 1 + 2 + \ldots + n = \Sigma[1..n]$. Ponieważ $\mathcal{S}_n$ jest sumą elementów ciągu arytmetycznego, nietrudno zauważyć, że

Fakt 1: $\mathcal{S}_n = n(n+1)/2$,

Fakt 2: $n \leqslant \mathcal{S}_n$.

Zanim przejdziemy do rozwiązania zadania, rozważmy następujący problem pomocniczy: dane są liczby n i S, znaleźć zbiór $Z \subseteq [1..n]$ taki, że $\Sigma Z = S$, o ile taki zbiór istnieje. Po pierwsze zauważmy, że:

Twierdzenie 3: $\Sigma^*[1..n] = [0..\mathcal{S}_n]$.

Dowód: Dowodzimy przez indukcję względem n. Dla $n = 1$ po lewej stronie równości mamy

$$\Sigma^*[1..1] = \Sigma^*\{1\} = \{0, 1\} = [0..1],$$

natomiast po prawej

$$[0..\mathcal{S}_1] = [0..1].$$

Przypuśćmy teraz, że twierdzenie jest prawdziwe dla $n = k$. Pokażemy, że jest ono prawdziwe także dla $n = k + 1$. Po pierwsze wiadomo, że największym elementem zbioru $\Sigma^*[1..(k+1)]$ jest $\mathcal{S}_{k+1}$. Pozostaje tylko udowodnić, że wszystkie liczby naturalne mniejsze od $\mathcal{S}_{k+1}$ też należą do $\Sigma^*[1..(k+1)]$. Weźmy zatem dowolną liczbę m ze zbioru $[1..\mathcal{S}_{k+1}]$. Mamy następujące przypadki:

- $m \leqslant k$; w tym przypadku na pewno $m \leqslant \mathcal{S}_k$ (fakt 2), czyli $m \in [0..\mathcal{S}_k]$. Stąd na mocy założenia indukcyjnego wnioskujemy, że $m \in \Sigma^*[1..k]$, więc także $m \in \Sigma^*[1..(k+1)]$.
- $m > k$, czyli $m \geqslant k + 1$; w takim przypadku m można przedstawić jako sumę $m = m' + (k+1)$. Ponieważ $m \geqslant k+1$, to $m' \geqslant 0$. Ponieważ $m \leqslant \mathcal{S}_{k+1}$, to $m' \leqslant \mathcal{S}_k$. Ostatecznie $m' \in [0..\mathcal{S}_k]$. Z założenia indukcyjnego wiemy, że m' daje się przedstawić jako suma elementów pewnego podzbioru zbioru $[1..k]$. Wnioskujemy zatem, że m daje się przedstawić jako suma elementów pewnego podzbioru zbioru $[1..(k+1)]$ (do podzbioru zbioru [1..k] o sumie elementów równej m' trzeba dołożyć element $k+1$). Stąd $m \in \Sigma^*[1..(k+1)]$.

Powyższe stwierdzenia kończą dowód. □

Na podstawie dowodu twierdzenia można skonstruować algorytm, który dla danej liczby m ze zbioru $[0..\mathcal{S}_n]$ wypisuje podzbiór zbioru $[1..n]$ o sumie elementów równej m. Schemat algorytmu wygląda następująco:

```
procedure rozkład(m, n : integer);
begin
  if m ⩽ 1 then begin
    if m = 1 then wypisz(1)
  end
  else begin
    if m ⩾ n then begin
      wypisz(n);
      rozkład(m − n, n − 1)
    end
    else
      rozkład(m, n − 1)
  end
```

* Przyjmujemy, że $\Sigma\varnothing = 0$, gdzie $\varnothing$ oznacza zbiór pusty.

```
end
```

Uważny czytelnik dostrzeże, że powyższy algorytm wynika niemalże wprost z dowodu indukcyjnego: rozpatrywanie przypadków w dowodzie odpowiada instrukcjom **if then else**, natomiast powoływanie się na założenie indukcyjne odpowiada rekurencyjnemu wywołaniu procedury (dla danych o mniejszym rozmiarze). Czytelnik z pewnością zauważy także, że rekurencja w procedurze *rozkład* może być zastąpiona przez iterację. Iteracyjna wersja algorytmu wygląda następująco:

```
procedure rozkład(m, n : integer);
begin
  while m > 0 do begin
    if m ⩾ n then begin
      wypisz(n);
      m := m − n
    end;
    n := n − 1
  end
end
```

Zbiór $[1..n]$ może zawierać wiele podzbiorów o sumie elementów równej m (np. dla $[1..4]$ oraz $m = 5$ mamy $\Sigma\{1,4\} = \Sigma\{2,3\} = 5$). Nasz algorytm wyznacza jeden z wielu możliwych rozkładów liczby m.

Powróćmy teraz do problemu pomocniczego (znaleźć podzbiór zbioru $[1..n]$ o sumie elementów równej S). Korzystając z twierdzenia oraz skonstruowanego algorytmu można sformułować następujące rozwiązanie: jeżeli S jest spoza przedziału $[0..\mathcal{S}_n]$, to nie istnieje $Z \subseteq [1..n]$ o sumie elementów równej m. Jeżeli natomiast $m \in [0..\mathcal{S}_n]$, to Z można wyznaczyć za pomocą procedury *rozkład*.

Powyższy rezultat pozwala wreszcie zmierzyć się z zadaniem olimpijskim. Mamy znaleźć ciąg $a_1, \ldots, a_n$ taki, że $a_1 + \ldots + a_n = S$, $a_1 = 0$ oraz różnica pomiędzy kolejnymi wyrazami wynosi 1 lub -1. Oznaczmy przez b_k różnicę pomiędzy wyrazem k-tym i $(k+1)$-szym, tj. $b_k = a_{k+1} - a_k$. Wówczas

$$\begin{aligned} a_1 + a_2 + a_3 + \ldots + a_n &= 0 + b_1 + (b_1 + b_2) + \ldots + (b_1 + b_2 + \ldots + b_{n-1}) = \\ &= (n-1) \cdot b_1 + (n-2) \cdot b_2 + \ldots 1 \cdot b_{n-1}. \end{aligned}$$

Zadanie sprowadza się zatem do wyznaczenia wartości współczynników b_k (równych 1 lub -1), żeby zachodziła równość

$$S = (n-1) \cdot b_1 + (n-2) \cdot b_2 + \ldots + 1 \cdot b_{n-1}.$$

Spróbujemy teraz przekształcić tę równość w ten sposób, żeby po prawej stronie, zamiast sumy liczb z przedziału [1..(n-1)] ze współczynnikami ze zbioru $\{-1, 1\}$, otrzymać sumę ze współczynnikami ze zbioru $\{0, 1\}$, dla której wartości współczynników umiemy wyznaczyć przy pomocy omówionego wcześniej rozwiązania problemu pomocniczego. W tym celu do każdego b_k trzeba dodać jeden (otrzymujemy wtedy współczynniki ze zbioru $\{0, 2\}$), a następnie podzielić przez 2. Dodanie jedynki do b_k oznacza zwiększenie prawej strony równania o wartość wyrazu mnożonego przez b_k. Ponieważ współczynniki b_k stoją przy kolejnych liczbach z przedziału $[1..(n-1)]$, to „dodanie jedynki" do wszystkich b_k powoduje zwiększenie wartości prawej strony równania o $\mathcal{S}_{n-1}$. Stąd otrzymujemy:

$$\frac{S + \mathcal{S}_{n-1}}{2} = (n-1) \cdot \frac{b_1 + 1}{2} + (n-2) \cdot \frac{b_2 + 1}{2} + \ldots + 1 \cdot \frac{b_{n-1} + 1}{2} \tag{1}$$

Algorytm będący rozwiązaniem zadania powinien stwierdzić, posługując się rozwiązaniem zadania pomocniczego, czy istnieje podzbiór zbioru $[1..(n-1)]$ o sumie elementów $(S + \mathcal{S}_{n-1})/2$ i ew. wyznaczyć go. Jeśli taki podzbiór nie istnieje, to nie istnieje także ciąg spełniający warunki zadania. W przeciwnym przypadku zawartość podzbioru określa, które współczynniki w sumie (1) mają wartość 0, a które 1, czyli które współczynniki b_k są równe -1, a które 1.

Należy pamiętać, że w zadaniu pomocniczym zakładaliśmy, iż rozkładana na składniki suma ma wartość całkowitą. W rozwiązaniu zadania olimpijskiego warunek ten należy sprawdzić, gdyż może się zdarzyć, że $S + \mathcal{S}_{n-1}$ jest liczbą nieparzystą. Oczywiście w takim przypadku ciąg spełniający warunki zadania nie istnieje.

Pozostaje kwestia implementacji opisanego algorytmu, co jednak, ze względu na jego prostotę, nie powinno nastręczać trudności. Algorytm wzorcowy najpierw sprawdza parzystość liczby $S + \mathcal{S}_{n-1}$ oraz przynależenie liczby $(S + \mathcal{S}_{n-1})/2$ do przedziału $[0..\mathcal{S}_{n-1}]$ (wystarczy sprawdzić przynależenie S do przedziału $[-\mathcal{S}_{n-1}..\mathcal{S}_{n-1}]$), a następnie rozkłada ją na składniki ze zbioru $[1..(n-1)]$ za pomocą iteracyjnej wersji procedury *rozkład*, konstruując na bieżąco ciąg (a_n). Jeżeli kolejny element zbioru $[1..(n-1)]$ należy do podzbioru o sumie elementów $(S + \mathcal{S}_{n-1})/2$, to stojący przy nim we wzorze (1) współczynnik b_k jest równy 1, czyli kolejny wyraz ciągu (a_n) musi być o jeden większy od poprzedniego. W przeciwnym przypadku, na mocy analogicznego rozumowania, kolejny wyraz ciągu (a_n) musi być o jeden mniejszy od poprzedniego. Ponadto wiadomo, że $a_1 = 0$. Na załączonej dyskietce Czytelnik znajdzie program SUM.PAS będący implementacją opisanego algorytmu.

Złożoność obliczeniowa algorytmu jest liniowa względem n. Algorytm korzysta z pamięci o stałym rozmiarze, niezależnym od danych wejściowych.

OPIS TESTÓW

Do sprawdzenia rozwiązań zawodników użyto 13-tu testów.

- SUM0.IN — test z treści zadania.
- SUM1.IN, SUM2.IN — niewielkie testy poprawnościowe.
- SUM3A.IN — brak rozwiązania, $S + \mathcal{S}_{n-1}$ nieparzyste.
- SUM3B.IN — prosty test posiadający rozwiązanie (do zgrupowania).
- SUM4A.IN — brak rozwiązania, zbyt małe S.
- SUM4B.IN — prosty test posiadający rozwiązanie (do zgrupowania).
- SUM5.IN — przypadek graniczny ($n = 1$, $S = 0$).
- SUM6.IN–SUM8.IN — testy wydajnościowe (duże n i S).
- SUM9.IN — maksymalne S.
- SUM10.IN — minimalne S.

Dla dwóch z nich poprawną odpowiedzią był plik zawierający słowo „NIE”. By unkinąć punktowania rozwiązań, które zawsze wypisują „NIE” do pliku wyjściowego, zastosowano tzw. grupowanie testów, tj. punkty za pary testów SUM3A.IN i SUM3B.IN oraz SUM4A.IN i SUM4B.IN przyznawano tylko wtedy, gdy oba testy zostały zaliczone.

Krzysztof Diks — autor zadania, opracowanie
Adam Borowski — program wzorcowy

AB-słowa

Każdy niepusty ciąg, którego elementami są małe litery a i b, a także ciąg pusty nazywamy ab-słowem. Jeżeli $X = [x_1, \ldots, x_n]$ *jest ab-słowem, a* i, j *takimi dowolnymi liczbami całkowitymi, że* $1 \leqslant i \leqslant j \leqslant n$, *to przez* $X[i..j]$ *będziemy oznaczali podsłowo* X *składające się z kolejnych liter* $x_i, \ldots, x_j$. *Powiemy, że ab-słowo* $X = [x_1, \ldots, x_n]$ *jest* ładnie zbudowane, *jeżeli zawiera tyle samo liter a, co b i dla każdego* $i = 1, \ldots, n$ *podsłowo* $X[1..i]$ *zawiera co najmniej tyle samo liter a, co liter b.*

Podamy teraz indukcyjną definicję podobieństwa *ładnie zbudowanych ab-słów:*

- *każde dwa puste ab-słowa (tzn. nie zawierające żadnych liter) są podobne,*
- *dwa niepuste, ładnie zbudowane ab-słowa* $X = [x_1, \ldots, x_n]$ *i* $Y = [y_1, \ldots, y_m]$ *są podobne, jeżeli są tej samej długości* ($n = m$) *i jest spełniony jeden z następujących warunków:*
  - $x_1 = y_1$, $x_n = y_n$ *oraz* $X[2..n-1]$ *i* $Y[2..n-1]$ *są ładnie zbudowanymi ab-słowami*
  - *istnieje* i, $1 \leqslant i < n$, *takie, że słowa* $X[1..i]$, $X[i+1..n]$ *są ładnie zbudowanymi ab-słowami i*
    - $Y[1..i]$, $Y[i+1..n]$ *są ładnie zbudowanymi ab-słowami i* $X[1..i]$ *jest podobne do* $Y[1..i]$ *oraz* $X[i+1..n]$ *jest podobne do* $Y[i+1..n]$ *lub*
    - $Y[1..n-i]$, $Y[n-i+1..n]$ *są ładnie zbudowanymi ab-słowami i* $X[1..i]$ *jest podobne do* $Y[n-i+1..n]$ *oraz* $X[i+1..n]$ *jest podobne do* $Y[1..n-i]$.

Stopniem zróżnicowania *niepustego zbioru* S *ładnie zbudowanych ab-słów nazywamy największą liczbę ab-słów, które można wybrać z* S *tak, żeby żadne dwa wybrane słowa nie były do siebie podobne.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego ABS.IN elementy zbioru* S;
- *policzy stopień zróżnicowania zbioru* S;
- *zapisze wynik w pliku tekstowym ABS.OUT.*

Wejście

W pliku tekstowym ABS.IN są zapisane:

- *w pierwszym wierszu liczba* n *elementów zbioru* S, $1 \leqslant n \leqslant 1000$;
- *w kolejnych* n *wierszach elementy zbioru* S, *tj. ładnie zbudowane ab-słowa, po jednym w każdym wierszu; pierwsza litera każdego ab-słowa jest pierwszym symbolem w wierszu i między kolejnymi literami w słowie nie ma żadnych innych znaków; każde słowo ma długość co najmniej 1, a co najwyżej 200.*

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego ABS.OUT należy zapisać jedną liczbę całkowitą — stopień zróżnicowania S.

Przykład

Dla pliku wejściowego ABS.IN o zawartości:

```
3
aabaabbbab
abababaabb
abaaabbabb
```

poprawnym rozwiązaniem jest plik ABS.OUT o zawartości:

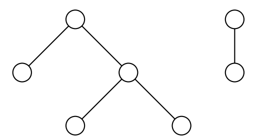
```
2
```

Twój program powienien szukać pliku ABS.IN w katalogu bieżącym i tworzyć plik ABS.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę ABS.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku ABS.EXE.

ROZWIĄZANIE

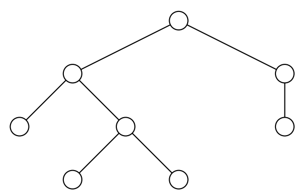
Łatwo zauważyć, że ab-słowo jest „ładnie zbudowane” wtedy i tylko wtedy, gdy po zamianie każdej litery a na nawias otwierający „(” i każdej litery b na nawias zamykający „)” otrzymamy poprawnie zbudowane **wyrażenie nawiasowe**. Każdemu poprawnie zbudowanemu wyrażeniu nawiasowemu odpowiada jednoznacznie ciąg drzew (z korzeniami) otrzymany w sposób zilustrowany na rysunku 1.

Rys. 1 Przykład odpowiedniości ab-słów, wyrażeń nawiasowych i ciągów drzew

$aabaababbbaabb =$ `(()(()()))(())` $=$ 

Podczepiając kolejne drzewa w ciągu do jednego, dodatkowego wierzchołka, otrzymamy drzewo jednoznacznie reprezentujące dane ab-słowo lub równoważne mu wyrażenie nawiasowe. Nowy wierzcholek ustanawiamy korzeniem drzewa. Drzewo powstałe z dodania korzenia do ciągu drzew z rysunku 1 przedstawiono na rysunku 2.

Rys. 2 Ciąg (dwóch) drzew z rysunku 1 połączonych w jedno drzewo nowym wierzchołkiem (aktualnym korzeniem)

$aabaababbbaabb =$ 

Powiemy, że dwa drzewa (posiadające wyróżnione korzenie) T_1 i T_2 są **izomorficzne** wtedy i tylko wtedy, gdy spełniony jest jeden z następujących warunków:

- Oba drzewa składają się tylko z pojedynczych wierzchołków.
- Jeśli $v_1, \ldots, v_k$ są synami korzenia drzewa T_1, a $u_1, \ldots, u_m$ są synami korzenia drzewa T_2, to $k = m$ (liczby synów korzeni w obu drzewach są takie same) i istnieje wzajemnie jednoznaczne przyporządkowanie synów korzenia T_1 synom korzenia T_2 takie, że poddrzewa zaczepione w odpowiadających sobie wierzchołkach są izomorficzne.

Na przykład drzewa odpowiadające wyrażeniom nawiasowym

```
( () (()) ((())) ((())()) )
( (()) ((())()) ((())) () )
( (()) (()(())) () ((())) )
```

są parami izomorficzne.

Nietrudno zauważyć, że relacja izomorfizmu drzew jest relacją:

- zwrotną (każde drzewo jest izomorficzne z samym sobą),
- symetryczną (jeśli drzewo T jest izomorficzne z drzewem T', to drzewo T' jest izomorficzne z drzewem T),
- przechodnią (jeśli drzewo T jest izomorficzne z drzewem T' i drzewo T' jest izomorficzne z drzewem T'', to drzewo T jest izomorficzne z drzewem T'').

O relacji, która jest zwrotna, symetryczna i przechodnia mówimy, że jest **relacją równoważności**.

Ponieważ ab-słowom jednoznacznie odpowiadają drzewa, to relację izomorfizmu drzew można w naturalny sposob przenieść na izomorfizm ab-słow: dwa ab-słowa są izomorficzne wtedy i tylko wtedy, gdy odpowiadające im drzewa są izomorficzne.

Analogicznie relacja podobieństwa na słowach zdefiniowana w treści zadania przenosi się w naturalny sposób na podobieństwa odpowiadających im drzew — podobieństwu podsłów odpowiada podobieństwo odpowiadających im ciagów poddrzew.

Niestety, relacja podobieństwa zdefiniowana w treści zadania nie jest relacją równoważności. Relacja podobieństwa ab-słów jest zwrotna, symetryczna, ale nie jest przechodnia, co, jak się okaże, bardzo utrudnia skonstruowanie efektywnego algorytmu dla naszego zadania.

Jeśli weźmiemy słowa (dla przejrzystości używamy nawiasów zamiast liter a i b)

$$A = (\ ()\ (())\ ((()))\ (((())))\ )$$
$$B = (\ (())\ (((())))\ ((()))\ ()\ )$$
$$C = (\ (())\ (((())))\ ()\ ((()))\ )$$

to A jest podobne do B, B jest podobne do C, ale A nie jest podobne do C!

Problem obliczania stopnia zróżnicowania danego zbioru ab-słów można wyrazić w terminach teorii grafów.

Powiemy, że wierzchołki grafu są **niezależne** wtedy i tylko wtedy, gdy nie są połączone krawędzią. Podzbiór wierzchołków grafu nazwiemy **zbiorem wierzchołków niezależnych** jeżeli wierzchołki w tym zbiorze są parami niezależne (tzn. jeżeli nie ma pomiędzy nimi żadnej krawędzi). Problem obliczania w grafie zbioru wierzchołków niezależnych o największym rozmiarze jest problemem bardzo trudnym obliczeniowo. Nie jest dla tego problemu znany żaden algorytm, który działałby w czasie wielomianowym, a istnienie takiego algorytmu jest bardzo mało prawdopodobne. Czytelnik, który znajdzie wielomianowy algorytm wyznaczający naliczniejszy zbiór wierzchołków niezależnych rozwiąże jeden z fundamentalnych problemów w informatyce, znany pod nazwą $P \stackrel{?}{=} NP$, zdobędzie sławę i pieniądze. Więcej na ten temat można znaleźć w [12].

Problem znajdowania stopnia zróżnicowania danego zbioru ab-słow można sprowadzić do obliczania rozmiaru najliczniejszego zbioru wierzchołków niezależnych w pewnym grafie. Tym grafem jest graf podobieństwa ab-słów, w którym wierzchołkami są dane ab-słowa, a dwa wierzchołki są połączone krawędzią wtedy i tylko, gdy odpowiadające im ab-słowa są podobne. Zauważmy, że gdyby relacją na słowach byla relacja izomorfizmu, to dla policzenia stopnia zróżnicowania zbioru ab-słów w tej relacji wystarczyłoby znaleźć liczbę spójnych składowych w grafie izomorfizmu, co daje się zrealizować w czasie wielomianowym.

Pan Adam Borowski pokazał, że istnieje wielomianowy algorytm, który dla dowolnego n-wierzchołkowego grafu G konstruuje n ab-słów, dla których graf podobieństwa jest właśnie grafem G. Stopień zróżnicowania takiego zbioru ab-słów jest zatem równy rozmiarowi najliczniejszego zbioru wierzchołków niezależnych w G. Wynika stąd, że gdybyśmy potrafili rozwiązać nasze zadanie w czasie wielomianowym, to rozwiązalibyśmy problem $P = NP$.*

Algorytm wzorcowy dla zadania ab-słowa działa w czasie wykładniczym. Na taką złożoność algorytmu ma wpływ sposób obliczania maksymalnego zbioru niezależnego w grafie podobieństwa. Zbiór ten jest obliczany przy pomocy metody zwanej **przeszukiwaniem z powrotami**, która w tym wypadku

* Pojawia się tu subtelna różnica polegająca na tym, że z rozwiązania wynika algorytm, który znajduje w czasie wielomianowym rozmiar najliczniejszego zbioru wierzchołków niezależnych, a nie sam zbiór. Dysponując jednakże takim algorytmem możemy w prosty sposób wyznaczyć w czasie wielomianowym również elementy tego zbioru. Konstrukcję takiego rozszerzenia algorytmu pozostawiamy jako ćwiczenie dla Czytelnika.

polega na generowaniu w systematyczny sposób wszystkich niezależnych podzbiorów i wybraniu z nich najliczniejszego. Taka metoda ma szanse powodzenia tylko wtedy, gdy graf, w którym szukamy najliczniejszego zbioru niezależnych wierzchołków, jest małego rozmiaru. Algorytm wzorcowy gwarantuje, że dla danych o rozmiarze wyspecyfikowanym w treści zadania, grafy, w których obliczane są zbiory niezależnych wierzchołków, są małych rozmiarów.

W jaki sposób to osiągamy? Nietrudno zauważyć, że dwa ab-słowa, które nie są izomorficzne, nie są do siebie podobne. Wynika stąd, że możemy zbiór wszystkich ab-słów podzielić na klasy, w których słowa są parami izomorficzne i dla każdej z nich policzyć oddzielnie stopień zróżnicowania. Poszukiwanym stopniem zróżnicowania całego zbioru jest suma stopni zróżnicowania poszczególnych klas. Algorytm podziału zbioru wszystkich ab-słów na klasy słów parami izomorficznych jest następujący:

- każdemu ab-słowu przyporządkowujemy sygnaturę, która jednoznacznie identyfikuje ab-słowa z tej samej klasy;
- sortujemy ab-słowa względem ich sygnatur;
- wyznaczamy klasy słów parami izomorficznych, jako ciągi kolejnych słów o takich samych sygnaturach.

Sygnaturę o potrzebnych nam własnościach można wybrać na wiele sposobów. Tutaj za sygnaturę przyjęliśmy drzewo ab-słowa przekształcone do **postaci kanonicznej** za pomocą permutowania synów każdego wierzchołka w tym drzewie.

Zdefiniujmy na drzewach następujący porządek liniowy (zob. [11]), oznaczany przez $\prec$:

(1) Jeżeli korzeń drzewa T ma mniej synów od korzenia drzewa T', to $T \prec T'$.

(2) Jeżeli dla uporządkowanych (w porządku $\prec$) ciągów synów korzeni T i T', odpowiednio $v_1, \ldots, v_k$ i $u_1, \ldots, u_k$, zachodzi $v_k = u_k, \ldots, v_{j+1} = u_{j+1}, v_j \prec u_j$ to $T \prec T'$.

Powiemy, że drzewo jest w **postaci kanonicznej**, jeśli synowie każdego wierzchołka w tym drzewie są uporządkowani nierosnąco zgodnie z porządkiem $\prec$ dla poddrzew, których są korzeniami.

Zdefiniujemy dodatkowo relację silnego podobieństwa, która pozwala na zmniejszenie rozmiaru grafu podobieństwa, w którym mamy znajdować najliczniejszy zbiór niezależnych wierzchołków.

Dwa ab-słowa A i B są **silnie podobne** wtedy i tylko wtedy, gdy dla każdego ab-słowa C, słowo A jest podobne do słowa C wtedy i tylko wtedy, gdy słowo B jest podobne do C.

Tak jak w przypadku podobieństwa, silne podobieństwo przenosi się w naturalny sposób na drzewa odpowiadające ab-słowom.

Łatwo sprawdzić, że wszystkie słowa, które są do siebie silnie podobne, są nierozróżnialne w sensie relacji podobieństwa — każdą grupę silnie podobnych słów możemy zastąpić jej jednym reprezentantem („skleić”), a stopień zróżnicowania zbioru słów nie zmieni się. Problem w tym, że nie dysponujemy szybkim algorytmem sprawdzającym silne podobieństwo słów. Spróbujemy jednak posłużyć się algorytmem, który czasami nie stwierdza silnego podobieństwa tam, gdzie ono zachodzi, ale za to jest szybki. Niedoskonałość algorytmu spowoduje, że po wykonaniu operacji sklejania pozostaną słowa, które są silnie podobne, ale nie zostały sklejone. No cóż, trudno. Okazuje się jednak, że zmniejszenie rozmiaru grafu, które dokonuje się w wyniku sklejania wierzchołków, jest wystarczające dla naszych celów.

Oto szybki algorytm „częściowego sprawdzania” silnego podobieństwa dwóch słów A i B:

- Jeśli liczba synów korzenia drzewa słowa A jest różna od liczby synów korzenia drzewa słowa B, to słowa nie są silnie podobne.

Załóżmy zatem, że liczby synów obu korzeni są takie same.

- Jeśli liczba synów jest mniejsza niż 4, synów korzeni drzew A i B sortujemy wg sygnatur zaczepionych w nich poddrzew. Jeśli odpowiadające sobie w tym porządku poddrzewa z obu drzewach A i B są silnie podobne, to słowa A i B też są silnie podobne.
- Jesli liczba synów jest większa od 3, sprawdzamy silne podobieństwo kolejnych (bez sortowania!) par poddrzew drzew słów A i B, których korzenie są synami korzeni tych drzew. Jeśli wszyskie pary poddrzew są silnie podobne, to drzewa A i B są też silnie podobne.

Oczywiście, jeśli słowa A i B są silnie podobne, to są podobne.

A oto wzorcowy algorytm obliczania stopnia zróżnicowania wejściowego zbioru ab-słów.

- Dzielimy dany zbiór ab-słów na klasy słów parami izomorficznych.
- Wewnątrz każdej klasy:
  - Znajdujemy zbiory słów silnie podobnych. Z każdego zbioru słów silnie podobnych usuwamy wszystkie słowa oprócz jednego.
  - Dla pozostałych słów budujemy graf podobieństwa.
  - W grafie podobieństwa znajdujemy największy zbiór wierzcholkow niezależnych.

Suma rozmiarów największych zbiorów wierzchołków niezależnych w każdej z klas słów parami izomorficznych jest poszukiwanym stopniem zróżnicowania wejściowego zbioru ab-słów.

TESTY

Do sprawdzania rozwiązań zawodników użyto 13-tu testów ABS0.IN–ABS12.IN. Test ABS0.IN był testem z treści zadania. W testach ABS3.IN, ABS5.IN, ABS6.IN, ABS7.IN, ABS11.IN relacja podobieństwa na zbiorze słów wejściowych pokrywała się z relacją izomorfizmu, przy czym testy ABS5.IN–ABS7.IN były testami o dużych rozmiarach (1000 słów o długościach dochodzących do 200 znaków). Pozostałe testy zaliczały tylko programy, których autorzy zauważyli, że relacja podobieństwa nie jest relacją rownoważności. W szczególności rozwiązanie dla zestawu danych ABS9.IN sprowadzało się do znalezienia najliczniejszego zbioru wierzchołków niezależnych w pewnym grafie ośmiowierzchołkowym.