Zawody III stopnia

opracowania zadaĆń

Marcin Kubica
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Układy asemblerowe

Firma Bajtel postanowiła usprawnić produkowane przez nią komputery, przez zastąpienie zaszytych w komputerach programów asemblerowych specjalnymi układami elektronicznymi, zwanymi układami asemblerowymi. Programy asemblerowe składają się wyłącznie z przypisań. Każde przypisanie jest określone przez cztery elementy:

- *dwa rejestry, z których pobierane są dane,*
- *operację elementarną, wykonywaną na tych danych,*
- *rejestr, w którym umieszczany jest wynik.*

Zakładamy, że jest co najwyżej 26 rejestrów oznaczonych małymi literami alfabetu angielskiego od `a` *do* `z` *oraz są co najwyżej 4 różne operacje elementarne oznaczone wielkimi literami angielskiego alfabetu od* `A` *do* `D`.

Układ asemblerowy ma:

- *wejścia oznaczone nazwami rejestrów, na każde wejście podawana jest wartość początkowa odpowiedniego rejestru, oraz*
- *wyjścia oznaczone również nazwami rejestrów, których wartości końcowe są kierowane do tych rejestrów.*

Wewnątrz układu znajdują się bramki. Każda bramka ma dwa wejścia i jedno wyjście. Wykonuje ona jedną operację elementarną na danych, które pojawiają się na jej wejściach i udostępnia wynik na swoim wyjściu. Wejścia bramek oraz wyjścia całego układu są połączone z wyjściami innych bramek lub wejściami układu. Wyjścia bramek oraz wejścia układu mogą być połączone z wieloma wejściami innych bramek lub wyjściami układu. Połączenia między bramkami nie tworzą cykli.

Układ asemblerowy jest równoważny programowi asemblerowemu, gdy dla dowolnego stanu początkowego rejestrów daje taki sam — jak program — stan końcowy rejestrów.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego UKL.IN opis programu asemblerowego;*
- *oblicza najmniejszą liczbę bramek, które musi zawierać układ asemblerowy równoważny z danym programem;*
- *zapisuje wynik w pliku tekstowym UKL.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego UKL.IN znajduje się jedna liczba całkowita n *(*$1 \leqslant n \leqslant 1000$*). Jest to liczba instrukcji programu.*

W n *kolejnych wierszach opisane są kolejne instrukcje programu. Każdy opis ma postać czteroliterowego słowa, zaczynającego się od symbolu operacji elementarnej:* `A`, `B`, `C` *albo* `D`. *Pozostałe trzy litery są małe. Druga i trzecia — to nazwy rejestrów, z których należy pobrać dane; czwarta — to nazwa rejestru, w którym należy umieścić wynik.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego UKL.OUT jedną liczbę całkowitą — minimalną liczbę bramek układu asemblerowego, równoważnego danemu programowi.

Przykład

Dla pliku tekstowego UKL.IN:

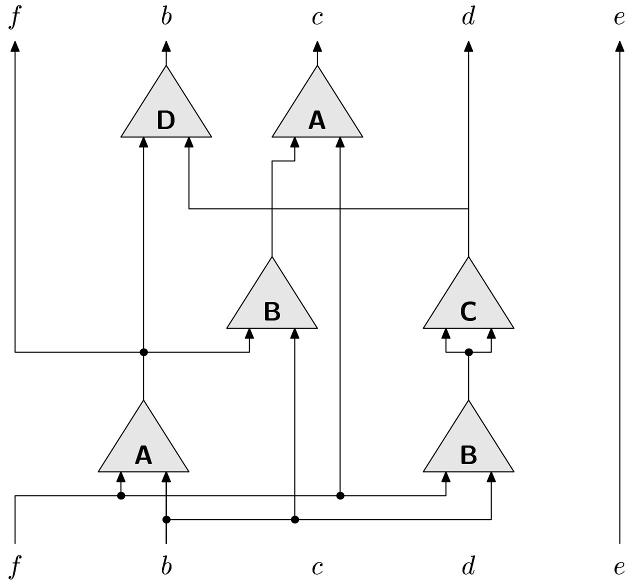
```
8
Afbc
```

```
Bfbd
Cddd
Bcbc
Afcc
Afbf
Cfbb
Dfdb
```

poprawnym rozwiązaniem jest plik wyjściowy UKL.OUT:

```
6
```

Układ równoważny danemu programowi przedstawiono na rysunku.



ROZWIĄZANIE

Zadanie to jest adaptacją problemu występującego przy optymalizacji kodu generowanego przez kompilatory, zwanego numerowaniem wartości (zob. [6]). Problem ten polega na takim przeorganizowaniu ciągu instrukcji, aby pozbyć się tych, które niepotrzebnie się powtarzają oraz tych, których wyniki są nieistotne.

Rozwiązanie polega na zbudowaniu grafu, którego wierzchołki reprezentują wartości pojawiające się w trakcie obliczeń. Są to wierzchołki reprezentujące początkowe wartości rejestrów oraz „bramki" — wierzchołki reprezentujące wyniki operacji elementarnych. Dla każdej bramki pamiętamy kod operacji elementarnej wykonywanej przez bramkę oraz dwie krawędzie prowadzące od wierzchołków reprezentujących argumenty operacji do bramki. Krawędzie prowadzące do bramki są pamiętane tak, aby było wiadomo, która reprezentuje pierwszy, a która drugi argument operacji.

Graf budujemy na bieżąco, podczas wczytywania danych. W danej chwili pamiętamy też, które wierzchołki reprezentują wartości rejestrów. Początkowo graf składa się wyłącznie z wierzchołków reprezentujących inicjalne wartości rejestrów. Dla każdej kolejnej instrukcji or_1r_2r sprawdzamy, czy w budowanym grafie istnieje bramka o kodzie operacji o, do której prowadzą krawędzie, odpowiednio z wierzchołków reprezentujących wartości rejestrów r_1 i r_2. Jeśli taka bramka nie istnieje, to dodajemy ją do grafu. Następnie przyjmujemy, że bramka ta reprezentuje wartość rejestru r. Zauważmy, że po utworzeniu grafu instrukcje dające te same wyniki są reprezentowane przez jedną bramkę. Jednocześnie wiadomo, które wierzchołki grafu reprezentują końcowe wartości rejestrów.

Tak utworzony graf może jednak zawierać bramki, które nie mają wpływu na końcowe wartości rejestrów. Aby wyznaczyć, które bramki są potrzebne, przechodzimy po krawędziach grafu wstecz, począwszy od wierzchołków reprezentujących końcowe wartości rejestrów. Jednocześnie zliczamy odwiedzane bramki. Bramki, które nie zostaną odwiedzone, nie mają wpływu na końcowe wartości rejestrów. Tak więc liczba odwiedzonych bramek jest szukanym wynikiem.

Dla efektywności algorytmu kluczowy jest sposób reprezentacji danych oraz sprawdzania, czy w grafie jest bramka o zadanym kodzie operacji, do której prowadzą krawędzie z zadanych wierzchołków. Najprostsza implementacja polega na każdorazowym przeglądaniu wszystkich bramek w grafie. Wówczas pierwsza faza algorytmu — budowa grafu — ma koszt $\Theta(n^2)$ (gdzie n, to liczba instrukcji). Lepszy koszt można uzyskać pamiętając opisy bramek (złożone z kodu operacji oraz wierzchołków reprezentujących argumenty) w słowniku o logarytmicznym czasie dostępu (np. w drzewie AVL, por. [9]). Wówczas koszt pierwszej fazy algorytmu wynosi $\Theta(n \log n)$. Implementacja takiej struktury danych jest jednak dosyć pracochłonna. Kolejną możliwością jest pamiętanie opisów bramek w tablicy mieszającej. W tym ostatnim przypadku, z uwagi na fakt, że liczba bramek jest stosunkowo niewielka, łatwo uzyskać koszt oczekiwany $\Theta(n)$.

Koszt drugiej fazy algorytmu, przy zastosowaniu np. przeszukiwania grafu w głąb, jest liniowy ze względu na liczbę wierzchołków w grafie. Tak więc koszt pierwszej fazy dominuje nad drugą.

Wśród rozwiązań przedstawionych przez zawodników kolejność opisanych faz była czasem odwrócona. Korzyścią płynącą z takiego rozwiązania jest obniżenie kosztu fazy utożsamiającej operacje, które mogą być reprezentowane przez te same bramki (jest to istotne, gdy ma ona koszt $\Theta(n^2)$). Jednak praktycznie oznacza to dwukrotną konstrukcję grafu. W rezultacie, gdy liczba instrukcji nie przekracza 1000, taka zamiana kolejności faz oznacza nieznaczne spowolnienie programu.

Okazuje się, że dla danych z takiego zakresu jak w treści zadania, wszystkie opisane rozwiązania działają praktycznie tak samo szybko. Biorąc pod uwagę to spostrzeżenie, a także fakt, że zadanie to było zadaniem rozgrzewkowym, przy ocenie rozwiązań przyjęto za punkt odniesienia rozwiązanie o koszcie $\Theta(n^2)$. Wszystkie sprawdzane rozwiązania, które uzyskały 100 pkt. działały w czasie 0,3–0,4 s, bez względu na rozmiar testów. Widać więc, że czas uruchomienia programu oraz wczytywania danych jest istotnie większy niż czas trwania obliczeń.

Przedstawimy teraz bardziej szczegółowo rozwiązanie o koszcie $\Theta(n^2)$. Konstruowany graf możemy reprezentować pamiętając opisy bramek (w tablicy *Bramki*). Każdy taki opis składa się z kodu operacji elementarnej `op` oraz opisów `a`, `b` krawędzi prowadzących do bramki. Krawędzie reprezentujemy za pomocą liczb całkowitych — liczby dodatnie oznaczają pozycje w tablicy bramek, od których prowadzą krawędzie, a liczby ujemne oznaczają początkowe wartości poszczególnych rejestrów. Podobnie reprezentujemy wartości rejestrów w trakcie konstrukcji grafu (w tablicy *Rejestry*). Biorąc pod uwagę przeszukiwanie grafu w drugiej fazie algorytmu, w opisie bramki pamiętamy również (pole *odw*), czy dana bramka została odwiedzona. Liczbę bramek pamiętamy na zmiennej *nb*.

```
const MAXN = 1000;
type
  TBramka = record
  op : char;
  a, b : integer;
  odw : boolean
end;
var
  Rejestry : array ['a'..'z'] of integer;
  Bramki : array [1..MAXN] of TBramka;
  nb : word;
```

Wyszukanie bramki o zadanym kodzie operacji elementarnej oraz węzłach, z których prowadzą do niej krawędzie, jest realizowane przez poniższą funkcję. Wynikiem funkcji jest pozycja szukanej bramki w tablicy *Bramki*, a jeśli takiej bramki nie ma, to wynikiem jest 0.

```
function ZnajdźBramkę(lop : char; la, lb : integer) : word;
var
  i : word;
begin
  for i := 1 to nb do with Bramki[i] do
```

```
    if (op = lop) and (a = la) and (b = lb) then begin
      ZnajdźBramkę := i;
      exit
    end;
  ZnajdźBramkę := 0
end;
```

Uwzględnienie kolejnej instrukcji w trakcie budowy grafu może być realizowane przez następującą prostą procedurę:

```
procedure PrzetwórzInstrukcję(lop, ra, rb, lr : char);
var i : word;
begin
  i := ZnajdźBramkę(lop, Rejestry[ra], Rejestry[rb]);
  if i = 0 then begin
    inc(nb);
    with Bramki[nb] do begin
      op := lop;
      a := Rejestry[ra];
      b := Rejestry[rb];
      odw := false
    end;
    i := nb
  end;
  Rejestry[lr] := i
end;
```

Poniżej przedstawiamy jeszcze funkcję realizującą drugą część algorytmu. Chcąc obliczyć wynik należy zsumować wyniki tej funkcji dla wszystkich wierzchołków grafu reprezentujących końcowe wartości rejestrów.

```
function Przejdź(i : integer) : word;
begin
  if (i ⩽ 0) or (Bramki[i].odw) then Przejdz := 0
  else with Bramki[i] do begin
    odw := true;
    Przejdź := 1 + Przejdz(a) + Przejdz(b)
  end
end
```

Pełną treść programu można znaleźć na załączonej dyskietce.

TESTY

W poniższej tabelce zestawiono liczbowe charakterystyki wszystkich testów. Instrukcje „zdublowane", to te, które są usuwane w pierwszej fazie algorytmu — w trakcie budowy grafu. Instrukcje „zbędne" to te, które nie są odwiedzane w drugiej fazie algorytmu.

	Liczba instrukcji			Liczba
Test	wszystkich	zdublowanych	zbędnych	bramek
UKL0.IN	8	1	1	6
UKL1.IN	10	1	3	6
UKL2.IN	20	3	8	9
UKL3.IN	48	24	0	24
UKL4.IN	100	0	99	1
UKL5.IN	6	1	2	3
UKL6.IN	100	0	0	100
UKL7.IN	6	2	3	1
UKL8.IN	200	55	14	131
UKL9.IN	1000	743	175	82
UKL10.IN	1000	498	0	502
UKL11.IN	1000	0	0	1000
UKL12.IN	6	1	1	4
UKL13.IN	1000	999	0	1
UKL14.IN	6	1	1	4

Aby wychwycić te programy, które zawsze dawały odpowiedź jeden, albo zawsze wypisywały na wyjście liczbę wszystkich instrukcji, zgrupowano niektóre testy z prostymi testami zawierającymi 6 instrukcji. Rozwiązanie uzyskiwało punkty za grupę testów tylko wtedy, gdy było poprawne dla wszystkich testów z grupy. I tak, zgrupowano testy: 4 i 5, 6 i 7, 11 i 12, oraz 13 i 14. Poniżej przedstawiamy krótką charakterystykę ciekawszych testów.

- UKL3.IN — 4 warstwy, po 6 bramek w każdej warstwie, każdej bramce odpowiadają dokładnie dwie instrukcje.
- UKL4.IN — w każdej ze 100-u instrukcji wynik jest umieszczany w tym samym rejestrze (`d`), który nie jest argumentem żadnej instrukcji (zatem wszystkie instrukcje, poza ostatnią, są zbędne).
- UKL6.IN — instrukcja `Ahhh` powtarza się 100 razy (szukany układ bramek przypomina drabinkę o 100-u szczeblach).

Piotr Chrząstowski-Wachtel
autor zadania, opracowanie

Adam Borowski
program wzorcowy

Bankomaty

Każdy członek Bajtlandzkiej Kasy Pożyczkowej ma prawo pożyczyć dowolną sumę mniejszą niż 10^{30} bajtlandzkich dukatów, ale musi ją w całości zwrócić do Kasy nie później niż po upływie 7 dni. W sali obsługi klientów Kasy ustawiono 100 bankomatów ponumerowanych od 0 do 99. Każdy bankomat wykonuje tylko jedną operację: wypłaca albo przyjmuje ustaloną kwotę. Bankomat o numerze i wypłaca 2^i dukatów, jeśli i jest parzyste, zaś przyjmuje 2^i dukatów, jeśli i jest nieparzyste. Gdy klient zamierza wypożyczyć ustaloną kwotę, trzeba zbadać, czy będzie mógł ją pobrać, korzystając co najwyżej raz z każdego z bankomatów i jeśli tak, wyznaczyć numery bankomatów, z których należy skorzystać. Trzeba również zbadać, czy będzie mógł ją zwrócić w podobny sposób i jeśli tak, wyznaczyć numery bankomatów, z których należy skorzystać w celu wykonania tej operacji.

Przykład

Klient, który zamierza pożyczyć 7 dukatów, pobiera najpierw 16 dukatów w bankomacie nr 4 i 1 dukata w bankomacie nr 0, a następnie oddaje 8 dukatów w bankomacie nr 3 i 2 dukaty w bankomacie nr 1. Żeby zwrócić pożyczoną kwotę 7 dukatów, pobiera najpierw 1 dukata w bankomacie nr 0, a następnie oddaje 8 dukatów w bankomacie nr 3.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego BAN.IN liczbę klientów n i dla każdego klienta wysokość kwoty, jaką zamierza pożyczyć w Kasie;*
- *dla każdego klienta sprawdza, czy będzie mógł pobrać ustaloną kwotę korzystając co najwyżej raz z każdego bankomatu i jeśli tak, wyznacza numery bankomatów, z których należy skorzystać, oraz czy będzie mógł ją zwrócić w podobny sposób i jeśli tak, wyznacza numery bankomatów, z których należy skorzystać w tym celu;*
- *zapisuje wyniki w pliku tekstowym BAN.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego BAN.IN znajduje się jedna liczba całkowita dodatnia $n \leqslant 1000$. Jest to liczba klientów.

W każdym z kolejnych n wierszy jest jedna liczba całkowita dodatnia, mniejsza niż 10^{30}, zapisana za pomocą co najwyżej 30 cyfr dziesiętnych.

Liczba w i-tym z tych wierszy to wysokość kwoty, którą zamierza pożyczyć klient nr i.

Wyjście

W każdym z 2n kolejnych wierszy pliku wyjściowego BAN.OUT należy zapisać malejący ciąg liczb całkowitych dodatnich z zakresu [0..99] oddzielonych pojedynczymi odstępami albo jedno słowo NIE:

- *w pierwszym wierszu i-tej pary wierszy — numery bankomatów, w porządku malejącym, z których powinien skorzystać klient numer i, by pobrać pożyczkę, albo słowo NIE, gdy nie może jej pobrać zgodnie z ustalonymi regułami;*
- *w drugim wierszu i-tej pary — numery bankomatów, w porządku malejącym, z których powinien skorzystać klient numer i, oddając pożyczkę, albo słowo NIE.*

Przykład

Dla pliku tekstowego BAN.IN:

```
2
7
```

```
63382530011411470074835160269
```

poprawnym rozwiązaniem jest plik wyjściowy BAN.OUT:

```
4 3 1 0
3 0
NIE
99 3 1
```

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

ROZWIĄZANIE

Zadanie o bankomatach prezentowało ciekawy system zapisywania liczb całkowitych: system o podstawie -2. System ten jest binarnym systemem pozycyjnym, czyli takim, że pozycja cyfry decyduje o związanej z nią wartości, a jedynymi dozwolonymi cyframi są 0 oraz 1, z tym że na $i-$tej pozycji od prawej w zapisie liczby występuje cyfra odpowiadająca wartości $(-2)^i$. Zatem bitom, licząc od prawej, odpowiadają wartości $1, -2, 4, -8, 16, -32, \ldots$. Pojawienie się w zapisie pozycyjnym liczby bitu 1 oznacza dodanie odpowiadającej mu wartości $(-2)^i$, a 0 — pominięcie jej. Kolejne bankomaty odpowiadały kolejnym pozycjom bitów, a wybór bankomatu albo jego pominięcie — jedynce lub zeru na tej pozycji.

Okazuje się, że w tym systemie można zapisać każdą liczbę całkowitą i to dokładnie na jeden sposób. (Rozważamy tylko najkrótszy możliwy zapis.) Kolejnych kilka liczb całkowitych w okolicy zera ma następujące reprezentacje:

-5	1111
-4	1100
-3	1101
-2	10
-1	11
0	0
1	1
2	110
3	111
4	100
5	101

System ten był badany na początku lat 60-tych jako alternatywny do układu dwójkowego w procesorach komputerowych. Miał on kilka zalet, np. ujednolicenie zapisu i działań na liczbach dodatnich i ujemnych, brak wyróżnionych bitów, a także kilka wad, np. niesymetryczność zakresu liczb dodatnich i ujemnych w n-bitowej reprezentacji. Wadą tego systemu był też sposób obliczania liczby przeciwnej do danej, co okazało się niebanalną operacją. Widać, że postać każdej liczby w zapisie o podstawie -2 oraz postać liczby przeciwnej do niej różnią się od siebie zasadniczo.

Zadanie o bankomatach polegało właśnie na znalezieniu reprezentacji bitowej zadanej liczby i liczby przeciwnej do niej w układzie o podstawie -2.

Algorytm przedstawienia dowolnej nieujemnej liczby x w układzie pozycyjnym o podstawie $b \geqslant 2$ jest następujący. W tablicy cyfr $C[0..\mathit{liczbacyfr}-1]$ **of** $0..b-1$ umieszczamy od końca kolejne cyfry przedstawienia:

```
for i:=0 to liczbacyfr − 1 do
  begin
  C[i] := x mod b;
    x := x div b
end
```

Dzielimy zatem liczbę x przez b tyle razy, ile cyfr przedstawienia sobie zażyczymy. W tablicy cyfr C od prawej do lewej odnotowujemy reszty z tego dzielenia. Jeżeli wartość x po kilku iteracjach spadnie do zera, to pozostała część tablicy, aż do lewego końca, zostanie uzupełniona zerami. Możemy też wyjść wcześniej z pętli, zmieniając jej nagłówek na **while** $x \neq 0$ **do**. W tej wersji trzeba albo wypełnić na początku tablicę C zerami, albo pamiętać, gdzie się zaczyna zapis liczby.

Powyższy algorytm dla $b = 2$ nazywany jest niekiedy algorytmem *zamiany liczby z układu dziesiętnego na dwójkowy*, co jest mylną nazwą, bo z układem dziesiętnym ma on niewiele wspólnego. Może tyle tylko, że do reprezentowania liczby jesteśmy przyzwyczajeni używać układu dziesiętnego. W algorytmie bowiem odwołujemy się do tego, co z liczby pozostanie w kolejnych przebiegach pętli algorytmu, a nie do rozwinięcia dziesiętnego tej liczby i jej pozostałości „wewnątrz” algorytmu.

Ponieważ mamy znaleźć przedstawienie liczb w układzie o podstawie -2, więc najprościej byłoby użyć tego algorytmu dla $b = -2$. Problem polega na tym, że nie jest jasne, jakie znaczenie nadać wyrażeniom x **mod** b oraz x **div** b, gdy b lub x są ujemne. Aby wyjaśnić tę kwestię można spróbować użyć dwóch metod: napisać prosty program w Pascalu i najzwyczajniej w świecie przyjrzeć się kilku wartościom, co oczywiście nie da stuprocentowej pewności co do poprawności, ale przynajmniej pozwoli się upewnić co do koncepcji. Drugi sposób to oczywiście spróbować udowodnić poprawność algorytmu dla ujemnych b.

Napisanie w Pascalu lub w C programu, który realizuje pierwszy pomysł dla $b = -2$ przynosi rozczarowanie: w tablicy C, np. dla $x = 11$, pojawiają się 3 wartości: $-1, 0, 1$ — zupełnie nie to, czego się spodziewaliśmy.

Spróbujmy zatem drugiej metody — dowodu poprawności algorytmu. Oczywiście możemy to potraktować w formie żartu, bo skoro pierwsza metoda zawiodła, to po co dalej rozmyślać? Nie jest to jednak do końca taki czczy żart, bo jeśli przyjrzymy się wynikom, to zobaczymy, że te rozwinięcia, które nie zawierają minus jedynek są dobre, a te, które zawierają — też są sensowne, tyle że sugerują wzięcie odpowiednich wartości z minusem. Na przykład liczba 11 ma rozwinięcie (-1)0(-1)1, czyli jakby $(-1) * (-8) + 0 * (4) + (-1) * (-2) + 1 * (1) = 11$. Coś się tu jednak zgadza. W innych przypadkach jest podobnie. Może zatem wystarczy się jakoś uwolnić od tych minus jedynek i w ogóle będzie dobrze? Wróćmy do pomysłu dowodu algorytmu zamiany podstawy układu pozycyjnego i zobaczmy, gdzie się nam dowód załamie.

Jaka jest interpretacja operacji **div** i **mod**? Oczywiście chodzi o iloraz i resztę dzielenia całkowitoliczbowego, czyli o takie dwie liczby d i r, że

$$x = db + r, \qquad \text{gdzie} \quad 0 \leqslant r < |b|. \tag{1}$$

Liczby d i r są wyznaczone jednoznacznie przez dowolne x oraz b, takie że $|b| > 1$.

Twierdzenie 1: Przy przyjętej interpretacji operacji **div** i **mod** algortym z poprzedniej strony jest poprawny dla każdej podstawy b takiej, że $|b| \geqslant 2$.

Dowód: Indukcja ze względu na $|x|$. Zacznijmy od przypadku $b \geqslant 2$ oraz $x \geqslant 0$. Najpierw odnotujmy, że liczba 0 daje właściwe przedstawienie przy każdej podstawie $b > 1$: wszystkie operacje **div** i **mod** dają w wyniku zero, więc tablica C zostanie wypełniona samymi zerami. Teraz załóżmy, że nasz algorytm działa poprawnie dla wszystkich wartości nieujemnych mniejszych niż pewne $x > 0$ (czyli wypełnia dla takich wartości tablicę C właściwymi cyframi). Udowodnimy, że w tym przypadku będzie poprawnie działał również dla liczby x.

Prześledźmy, co się stanie, gdy wykonamy pierwszy obrót pętli naszego algorytmu. Do tablicy C na pozycji 0 zostanie wpisany wynik x **mod** b, a liczba x stanie się równa x **div** b. Ponieważ $b > 1$, więc liczba x **div** b jest mniejsza niż x. Zatem na mocy założenia indukcyjnego liczba x **div** b zostanie w dalszej części algorytmu prawidłowo rozwinięta w komórkach $C[1], C[2], \ldots$, czyli do tablicy zostaną wpisane właściwe cyfry rozwinięcia $x\mathbf{div}b$ przy podstawie b, tyle że przesunięte o 1 w lewo. Ze względu na to, że przesunięcie o 1 w lewo oznacza w każdym potęgowym układzie pozycyjnym przemnożenie przez b, otrzymana liczba będzie równa

$$b \cdot (x \ \mathbf{div}\ b) + (x \ \mathbf{mod}\ b) = x,$$

co było do pokazania (na razie dla $x \geqslant 0$, $b \geqslant 2$).

Teraz najważniejsze odkrycie pozwalające nam przenieść ten dowód na ujemne wartości b oraz x. Zauważmy, że jedynym miejscem dowodu, w którym zakładaliśmy, że $b \geqslant 2$ i $x \geqslant 0$ było wykorzystanie kroku indukcyjnego: chodziło o to, żeby liczba x **div** b była mniejsza od x. Reszta rachunków nie wymagała ani dodatności b, ani nieujemności x.

Jak zatem zabrać się za ujemne wartości b i x? Wystarczy wymusić istotny postęp w „zmniejszaniu" się liczby x. W tym celu zauważmy, że dla $b \leqslant -2$, dla każdego x, poza jednym przypadkiem, wartość $|x$ **div** $b|$ jest mniejsza, niż $|x|$. Tym jedynym przypadkiem jest $x = -1$. Wtedy jednak x **div** $b = 1$ i w zasadzie można na tym poprzestać, bo w kolejnym kroku 1 **div** $b = 0$. Czyli poza -1, zarówno dla dodatnich, jak i dla ujemnych liczb x, mamy $|x$ **div** $b| < |x|$. Ze względu na to, że również dla -1 w następnym kroku osiągniemy zero uzyskując właściwą reprezentację: $-1 = (0\cdots01(b-1))_b$, rozumowanie możemy tu zakończyć. Formalnie powiększamy bazę indukcji o przypadki $x = -1$ i $x = 1$. Następnie zakładamy, że teza zachodzi dla wszystkich liczb co do modułu mniejszych od $|x|$, po czym przeprowadzamy wnioskowanie identyczne z poprzednim, które przekona nas, że algorytm działa poprawnie dla każdej liczby całkowitej x i dla każdej liczby b o module większym od 1. □

Zaraz, zaraz! To dlaczego nasz program w Pascalu dał złe wyniki? Skąd się wzięły te minus jedynki? Ja tego szczerze powiedziawszy nie rozumiem. To znaczy nie rozumiem, dlaczego tak wybitny informatyk, jakim jest Niklaus Wirth, projektując Pascala, w tak dziwaczny sposób zdefiniował operacje **div** i **mod**. Okazuje się, że dla liczb ujemnych pascalowe odpowiedniki wyników operacji **mod** i **div** nie mają nic wspólnego ze wzorem (1). Na przykład, zgodnie ze wzorem (1), (-7) **div** $(-2) = -4$, (-7) **mod** $(-2) = 1$, podczas gdy w Pascalu odpowiednie wyniki, to 3 i -1.

Widać więc, że przy nieograniczonej liczbie bankomatów każdą sumę dukatów da się pożyczyć i oddać, a algorytm wyboru bankomatów jest bardzo prosty. Jego istotę (z pominięciem problemów związanych z realizacją omawianych operacji) oddaje następujący kod poprawny dla każdej podstawy b:

```
for i:=0 to LiczbaBankomatow − 1 do
  begin
    Bankomat[i] := x mod (abs(b)); {u nas b = −2 }
    x := (x − Bankomat[i]) div b {odejmujemy, żeby uniknąć niejasności
                                                          z operacją div}
  end
```

Rzecz jasna, gdyby po zakończeniu pętli wartość x była różna od zera, to oznaczałoby to, że nie starcza nam bankomatów na wyrażenie aż tak dużej co do modułu liczby i byłaby to jedyna przyczyna, dla której odpowiedzią powinno być NIE. W rzeczywistości ograniczenia w zadaniu były tak dobrane, żeby na końcach przedziału występowały wartości niewyrażalne na 100 pozycjach w układzie o podstawie (-2).

Tym razem obowiązek zrealizowania operacji **div** i **mod** spoczywał na zawodnikach, bowiem zakres liczb przekraczał rozmiar wszelkich standardowych typów, w których te operacje są zdefiniowane na poziomie języka (zresztą, o czym była mowa, niezbyt szczęśliwie z punktu widzenia naszych potrzeb). Nie było to trudne, bo pierwsza operacja wymaga sprawdzenia jednego bitu, a druga — generując na przemian wyniki dodatnie i ujemne — sprowadzała się do ucinania ostatniej cyfry z ewentualnym odjęciem jedynki od wyniku dla argumentu ujemnego.

TESTY

Testy były podzielone na 11 grup.

- BAN0.IN — test z treści zadania.
- BAN1.IN — jedna liczba równa 1.
- BAN2.IN — 3 niewielkie liczby jednobajtowe.
- BAN3.IN — 100 liczb mieszczących się w typie **longint** (4 bajty).
- BAN4.IN — wszystkie liczby od 1 do 1000.
- BAN5.IN — różne liczby, w tym kilka trudnych, na granicy reprezentowalności.

- BAN6.IN — 442 liczby postaci $2^i, 2^i + 1, 2^i - 1, F(i)$, gdzie $F(i)$, to i-ta liczba Fibonacciego.
- BAN7.IN — 978 losowych liczb o liczbie cyfr z przedziału 1..30.
- BAN8.IN — 1000 losowych liczb o maksymalnej długości 30 cyfr.
- BAN9.IN — 1000 największych liczb trzydziestocyfrowych cyfr (harmonia dziewiątek z różnymi końcówkami).
- BAN10.IN — 1000 losowych liczb 28–30-cyfrowych

Adam Borowski
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Gonitwa

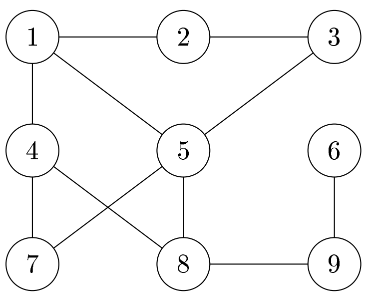
Gonitwa jest grą planszową dla dwóch osób A i B. Plansza do gry składa się z pól ponumerowanych kolejnymi liczbami naturalnymi, poczynając od 1. Dla każdej pary różnych pól wiadomo, czy są sąsiednie, czy nie. Każdy z graczy dysponuje jednym pionem, który początkowo znajduje się na wskazanym z góry polu planszy, każdy pion na innym polu. Ruch gracza polega na przesunięciu własnego piona na jedno z sąsiednich pól lub pozostawieniu piona na miejscu. Plansza ma następujące właściwości:

- *nie zawiera trójkątów, tzn. nie istnieją trzy różne pola, takie że każde dwa z nich są sąsiednie,*
- *każde pole może być osiągnięte zarówno przez gracza A, jak i przez gracza B.*

Gra składa się z wielu rund następujących po sobie. W jednej rundzie każdy z graczy wykonuje jeden ruch, przy czym gracz A zawsze wykonuje swój ruch przed ruchem gracza B.

Powiemy, że gracz B dogonił gracza A, jeżeli oba piony znajdą się na tym samym polu. Rozstrzygnij, czy dla danych początkowych położeń obu pionów, gracz B może dogonić gracza A, niezależnie od tego jak dobrze gra gracz A. Jeśli tak, to po ilu rundach gracz B dogoni gracza A, jeżeli gracz A stara się uciekać jak najdłużej, a gracz B stara się dogonić gracza A jak najszybciej i obaj grają optymalnie?

Przykład



Rozważmy planszę przedstawioną na rysunku. Sąsiednie pola planszy są połączone odcinkami. Jeżeli na początku gry pion gracza A znajduje się na polu 9, natomiast pion gracza B jest na polu 4, to gracz B dogoni A w trzeciej rundzie, o ile obaj gracze grają optymalnie. Gdyby pion gracza A znajdował się początkowo na polu 8, to ruszając z pola 4 gracz B nie ma szans na dogonienie A, jeżeli tylko ten gra optymalnie.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego GON.IN opis planszy i numery pól, na których początkowo umieszczone są piony graczy;*
- *sprawdza, czy gracz B może dogonić gracza A i jeżeli tak, to oblicza liczbę rund, po których B dogoni A przy założeniu, że obaj gracze grają optymalnie;*
- *zapisuje wynik w pliku tekstowym GON.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego GON.IN znajdują się cztery liczby całkowite n, m, a oraz b, oddzielone pojedynczymi odstępami, gdzie: $2 \leqslant n \leqslant 3000$, $n-1 \leqslant m \leqslant 15000$, $1 \leqslant a, b \leqslant n$ oraz $a \leqslant b$. Są to odpowiednio: liczba pól na planszy, liczba wszystkich różnych par (nieuporządkowanych) tych pól, które ze sobą sąsiadują, numer pola, na którym jest umieszczony pion gracza A oraz numer pola, na którym jest umieszczony pion gracza B.

W każdym z następnych m wierszy znajdują się dwie różne dodatnie liczby całkowite oddzielone pojedynczym odstępem. Liczby w każdym z tych wierszy są numerami dwóch pól, które ze sobą sąsiadują.

WYJŚCIE

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku GON.OUT jedno słowo NIE, jeśli gracz B nigdy nie dogoni gracza A, a w przeciwnym przypadku jedną liczbę całkowitą — liczbę rund, po których, gracz B dogoni gracza A.

PRZYKŁAD

Dla pliku wejściowego GON.IN:

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

poprawnym rozwiązaniem jest plik wyjściowy GON.OUT:

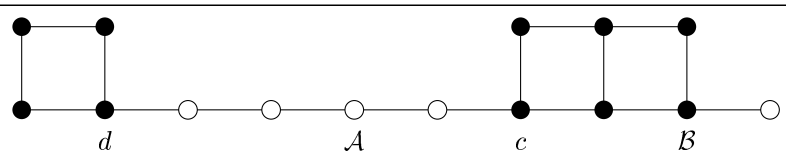
```
3
```

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

ROZWIĄZANIE

Dość oczywiste jest spostrzeżenie, że gracz $\mathcal{A}$ może uciekać nieskończenie długo tylko wtedy, jeśli uda mu się dotrzeć do grupy pól tworzących zamkniętą pętlę. Takich grup może być wiele, także jedno pole może należeć do wielu pętli. Pole należące do jakiejkolwiek pętli będziemy nazywać **bezpiecznym**.

Rys. 1 Przykładowa plansza z zaznaczonymi (przez zaczernienie) polami bezpiecznymi.



Kiedy gracz $\mathcal{A}$ dotrze do jakiegokolwiek z pól bezpiecznych może od tej chwili poruszać się po dowolnej z pętli przechodzących przez to pole, zmieniając ewentualnie kierunek ruchu na przeciwny. Jedyną strategią dla gracza $\mathcal{B}$ jest zatem niedopuszczenie, by $\mathcal{A}$ dotarł do któregoś z pól bezpiecznych. W tym miejscu łatwo zrobić błąd: jeśli gracz $\mathcal{A}$ dotrze do pola bezpiecznego, do którego **w tej samej rundzie** może dotrzeć też gracz $\mathcal{B}$, gra zakończy się wygraną gracza $\mathcal{B}$! Przykładem takiego pola jest pole c na rysunku 1.

Nieskończona trasa ucieczki istnieje wtedy i tylko wtedy, gdy na planszy istnieje pole bezpieczne x takie, że gracz $\mathcal{A}$ ma do pola x bliżej niż gracz $\mathcal{B}$. Możemy to sprawdzić znajdując odległości wszystkich pól zarówno od pola, z którego rozpoczyna grę gracz $\mathcal{A}$, jak i pola startowego gracza $\mathcal{B}$. Robimy to korzystając z algorytmu przeszukiwania grafu planszy wszerz (zob. [9]):

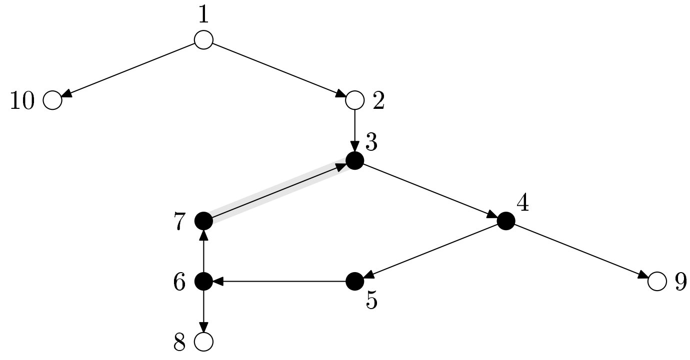
- wiemy, że jedynym polem odległym od $\mathcal{A}$ (odp. $\mathcal{B}$) o 0 jest jego pole startowe,

- dla każdego $k = 1, 2, 3, \ldots$ sprawdzamy, czy któreś z pól odległych od $\mathcal{A}$ (odp. $\mathcal{B}$) o $k-1$ ma jeszcze nie odwiedzonego sąsiada. Jeśli tak, zapamiętujemy że odległość tego sąsiada od $\mathcal{A}$ (odp. $\mathcal{B}$) wynosi k oraz dodajemy go do kolejki pól do sprawdzenia w kolejnym ruchu,
- obliczenia kończymy, gdy kolejka pól do sprawdzenia będzie pusta.

Teraz wystarczy sprawdzić, czy któreś z pól, którego odległość od $\mathcal{A}$ jest mniejsza niż odległość od $\mathcal{B}$, jest bezpieczne. I to jest właśnie najtrudniejsza część zadania: określenie, które pola są bezpieczne, a które nie.

Najbardziej efektywnym sposobem jest wykorzystanie przeszukiwania grafu w głąb (zob. [9] lub [12]) począwszy od dowolnego z wierzchołków grafu. Odwiedzanym polom nadajemy kolejne numery, a procedura przeszukująca wywołana rekurencyjnie zwraca najmniejszy numer pola, do którego można dojść idąc w danym kierunku i kończąc na polu, które było już odwiedzone lub z którego nie ma wyjścia.

Rys. 2 Przykładowy przebieg przeszukiwania w głąb, mającego na celu wyznacznie pól bezpiecznych. Numery przy wierzchołkach oznaczają kolejność odwiedzin, strzałki symbolizują kierunek przechodzenia krawędzi grafu.



Jak widać z przykładu (zob. rysunek 2), jeśli dane pole nie jest bezpieczne, zwrócony przez procedurę rekurencyjną numer musi być większy od numeru wierzchołka, z którego procedura ta była wywołana. Z drugiej strony, jeśli zwrócony numer jest mniejszy lub równy numerowi aktualnego wierzchołka, oznacza to, że jesteśmy na polu bezpiecznym.

Jeśli nie istnieje nieskończona trasa ucieczki, wynikiem zadania jest odległość od początkowej pozycji gracza $\mathcal{B}$ do najdalszego pola, którego odległość od początkowej pozycji gracza $\mathcal{A}$ jest nie większa niż odległość od $\mathcal{B}$. Najlepszą strategią dla gracza $\mathcal{A}$ jest umieszczenie piona na tym polu i czekanie, aż $\mathcal{B}$ tam dotrze.

Program wzorcowy najpierw wyznacza pola bezpieczne. Wierzchołki nie są numerowane kolejnymi liczbami, lecz liczbami oznaczającymi, na którym poziomie rekursji w przeszukiwaniu w głąb dany wierzchołek został znaleziony. Pomimo tej różnicy program daje poprawne wyniki. Uzasadnienie tego faktu pozostawiamy Czytelnikowi. Następnie wyznaczana jest tablica odległości wszystkich pól od początkowego pola gracza $\mathcal{A}$, oraz taka sama tablica z odległościami od początkowego pola gracza $\mathcal{B}$. Teraz mozemy już sprawdzić, czy istnieje pole bezpieczne leżące bliżej $\mathcal{A}$ niż $\mathcal{B}$. Jeśli tak, wypisujemy odpowiedź 'NIE', w przeciwnym przypadku określamy, na którym polu gracz $\mathcal{A}$ może się najdłużej chować. Złożoność powyższego algorytmu jest liniowa ze względu na rozmiar grafu.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 21 testów. Test GON0.IN był testem z treści zadania. W celu eleminacji przyznawania punktów rozwiązaniom wypisującym zawsze 'NIE' zgrupowano każdy test o numerze $2i-1$ z testem o numerze $2i$, dla $i = 1, \ldots, 10$. Testy GON1.IN–GON8.IN sprawdzały głównie poprawność programów, natomiast testy GON9.IN–GON20.IN były testami wydajnościowymi.

Wojciech Rytter
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Najlżejszy język

Dany jest alfabet A_k złożony z k początkowych liter alfabetu angielskiego. Każda litera w tym alfabecie ma określoną wagę, która jest dodatnią liczbą całkowitą. Wagą słowa zbudowanego z liter alfabetu A_k nazywamy sumę wag wszystkich jego liter. Językiem nad alfabetem A_k nazywamy dowolny skończony zbiór różnych słów utworzonych z liter tego alfabetu. Wagą języka nazywamy sumę wag wszystkich jego słów.

Mówimy że język jest **bezprefiksowy**, *gdy żadne jego słowo nie jest prefiksem (początkiem) jego innego słowa.*

Chcemy ustalić, jaka może być najmniejsza waga n-elementowego, bezprefiksowego języka nad alfabetem A_k.

Przykład

Załóżmy, że $k = 2$ waga litery a — $W(a) = 2$ oraz waga litery b — $W(b) = 5$.

Wtedy: waga słowa ab — $W(ab) = 2 + 5 = 7$. $W(aba) = 2 + 5 + 2 = 9$.

Waga języka $J = \{ab,\ aba,\ b\}$ — $W(J) = 21$.

Język J nie jest bezprefiksowy, ponieważ jego słowo ab jest prefiksem słowa aba.

Najlżejszym trzyelementowym bezprefiksowym językiem nad alfabetem A_2, o określonych powyżej wagach liter, jest $\{b,\ aa,\ ab\}$; jego waga wynosi 16.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego NAJ.IN dwie liczby naturalne n oraz k, a następnie wagi wszystkich k liter alfabetu A_k;*
- *oblicza najmniejszą wagę bezprefiksowego n-elementowego języka nad alfabetem A_k;*
- *zapisuje wynik w pliku tekstowym NAJ.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego NAJ.IN znajdują się dwie dodatnie liczby całkowite n i k oddzielone pojedynczym odstępem, $2 \leqslant n \leqslant 10000$, $2 \leqslant k \leqslant 26$. Są to odpowiednio: liczba słów języka i liczba liter alfabetu.

Drugi wiersz zawiera k dodatnich liczb całkowitych nie większych niż 10000, oddzielonych pojedynczymi odstępami — i-ta liczba jest wagą i-tej litery.

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego NAJ.OUT Twój program powinien wypisać jedną liczbę całkowita — wagę najlżejszego, bezprefiksowego, n-elementowego języka nad alfabetem A_k.

Przykład

Dla pliku wejściowego NAJ.IN:

```
3 2
2 5
```

poprawnym rozwiązaniem jest plik wyjściowy NAJ.OUT:

```
16
```

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

ROZWIĄZANIE

Na początek sformułujemy zadanie i algorytm w terminach drzew. Każdy bezprefiksowy język L na alfabetem A odpowiada drzewu, którego krawędzie są etykietowane literami z A. Każdemu prefiksowi słowa języka L odpowiada dokładnie jeden wierzchołek drzewa. Wagą krawędzi drzewa jest waga litery, będącej etykietą tej krawędzi.

Tak jak to jest przeważnie w informatyce, nasze drzewa rosną do góry nogami (korzeniami). **Wierzchołkiem zewnętrznym** (lub **liściem**) drzewa nazywamy wierzchołek, który nie ma wierzchołków potomnych (dzieci). **Wierzchołkiem wewnętrznym** nazywamy każdy wierzchołek drzewa, który nie jest wierzchołkiem zewnętrznym.

Ścieżką w drzewie nazywamy dowolny ciąg kolejnych krawędzi tego drzewa w kierunku od korzenia do pewnego liścia. Ścieżka nie musi kończyć się w liściu. Przez wagę ścieżki będziemy rozumieć sumę wag krawędzi leżących na tej ścieżce. Wagą całego drzewa nazwiemy sumę wag wszystkich ścieżek prowadzących do liści. Czasami zamiast mówić „waga ścieżki prowadzącej do wierzchołka v” będziemy mówili po prostu „waga wierzchołka v”.

Zadanie można teraz sformułować następująco: policz minimalną wagę drzewa, które ma n liści. Narysujmy wierzchołki drzewa tak, żeby każdy wierzchołek leżał na głębokości odpowiadającej jego wadze (korzeń drzewa na poziomie 0), tak jak na rysunkach 2 i 3.

Rozpatrzmy następujący przykład. Niech $n = 10$, $k = 4$ oraz niech wagi liter wynoszą:

$$W(a) = 2, \qquad W(b) = 3, \qquad W(c) = 8, \qquad W(d) = 9.$$

Wtedy ostatnie drzewo z rysunku 3 jest drzewem o minimalnym koszcie (być może nie jest to jedyne takie drzewo). Język

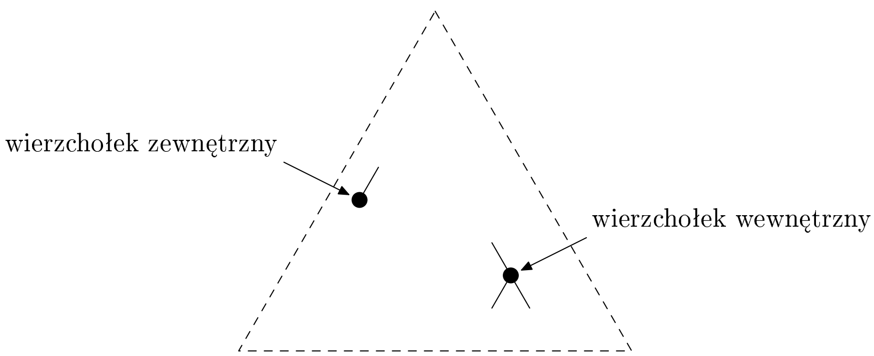
$$L = \{aaa,\ aab,\ aba,\ abb,\ ac, baa,\ bab,\ bb,\ c,\ d\}$$

jest 10-elementowym bezprefiksowym językiem o minimalnej wadze. Język ten składa się ze słów, którego słowa powstają ze ścieżek ostatniego drzewa na rysunku 3 (słowo otrzymujemy czytając po kolei etykiety krawędzi na ścieżce). Podstawową (oczywistą) własnością drzewa o minimalnej wadze jest to, że:

- liść nie może być na wyższym poziomie (mieć mniejszą wagę) niż jakikolwiek węzeł wewnętrzny.

W związku z powyższym **nie może** zajść sytuacja taka jak na rysunku 1. Jeśli mamy węzeł wewnętrzny o wadze w, to nie ma liścia (wierzchołka zewnętrznego) o wadze mniejszej niż w. Zatem w drzewie nie ma ”dziur”.

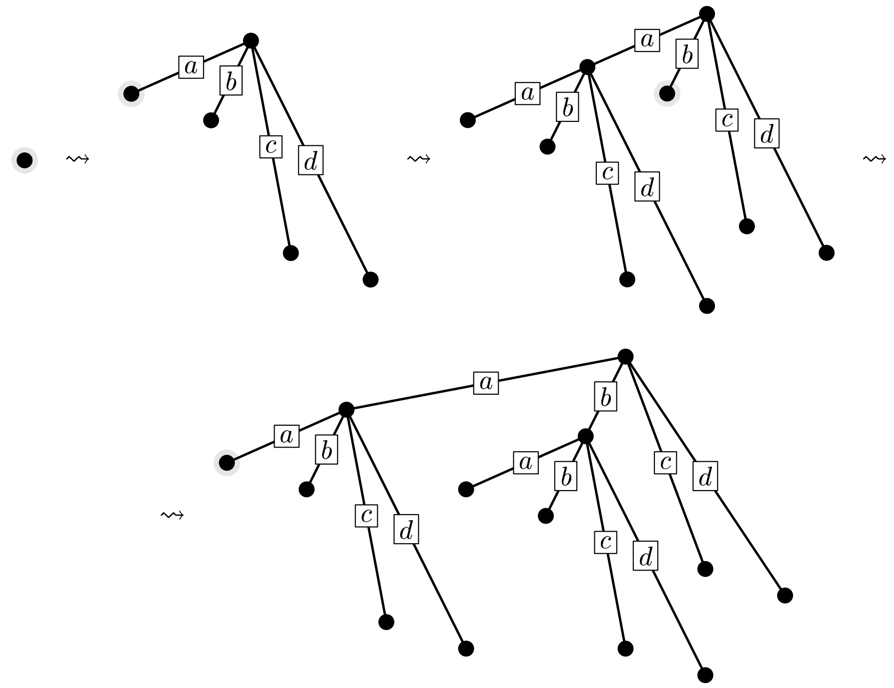
Rys. 1 Sytuacja, która nie może zajść w drzewie o minimalnej wadze.



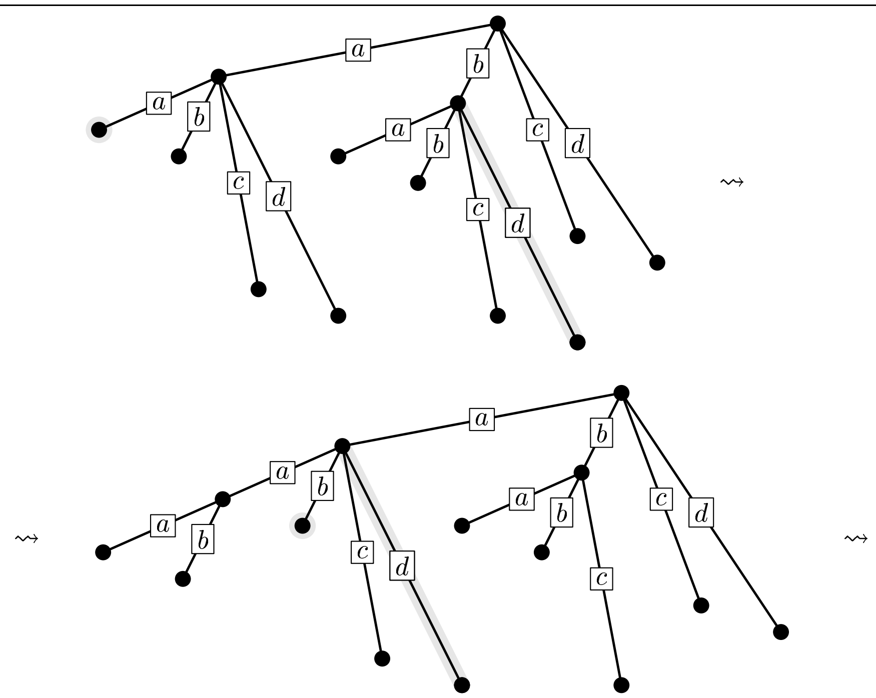
Powyższa własność implikuje następujący schemat algorytmu. Budujemy drzewo poziomami z góry na dół (korzeń jest na górze). Za każdym razem staramy się **rozwinąć** liść na najwyższym poziomie. Operację rozwijania kontynuujemy do momentu, gdy mamy n liści i próba rozwinięcia liścia nie zmniejsza wagi drzewa.

Oznaczmy początkowe k liter przez $a_1, a_2, \ldots, a_k$. Niech operacja *NowySyn(v, i)* oznacza utworzenie nowego węzła v', podczepionego do węzła v krawędzią o etykiecie a_i. Podstawową operacją jest następująca procedura:

Rys. 2 Pierwsze 3 iteracje algorytmu, rozwijamy „do pełna” 3 najlżejsze liście (zaznaczone na szaro) rozpoczynając od drzewa jednowierzchołkowego.

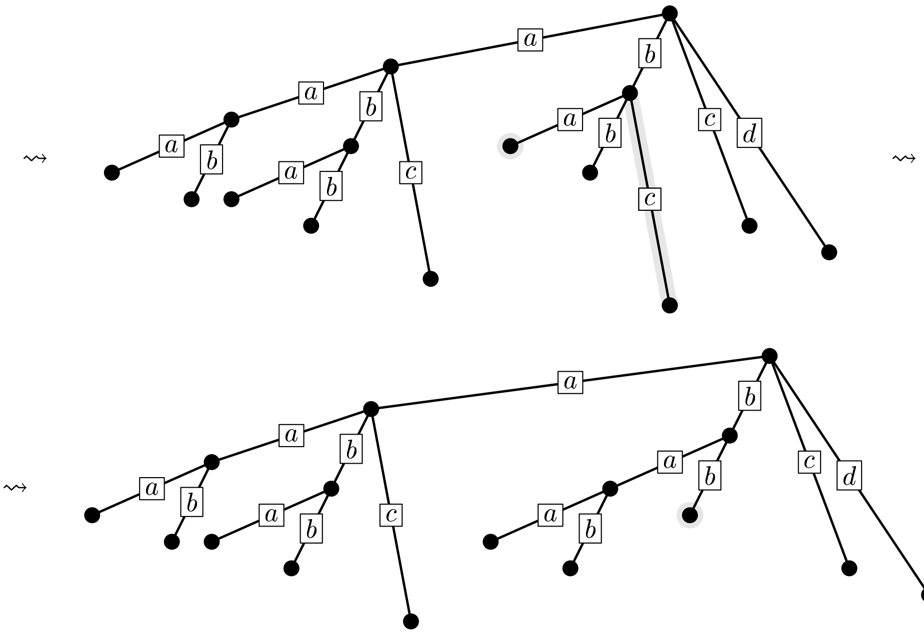


Rys. 3 Następne 3 iteracje, za każdym razem rozwijamy liść o najmniejszej wadze (zaznaczony na szaro) i usuwamy inne liście (krawędzie do usuwanych liści są zaznaczone na szaro). Pierwsze drzewo na tym rysunku jest ostatnim drzewem z rysunku 2.



procedure *RozwińLiść*(v : *wierzchołek*);
begin
  NowySyn(v, 1);

Rys. 3 dokończenie



for $i = 2$ **to** k **do**
if $liczbaLiści < n$ **then** $NowySyn(v,\ i)$[[[czy coś źle?]]]
else
    if istnieje liść w tż. $W(w) > W(v) + W(a_i)$
    then begin
        $Usuń(w)$;
        $NowySyn(v,\ i)$
    end
end

Teraz możemy sformułować algorytm:

$waga := +\infty$;
$nowaWaga := 0$;
$drzewo := \varepsilon$; { sam korzeń }
while $(waga > nowaWaga)$ **or** $(liczbaLiści < n)$ **do begin**
    $waga := nowaWaga$;
    $v :=$ liść o minimalnej wadze;
    $RozwińLiść(v)$;
    $nowaWaga :=$ aktualna waga drzewa
end

Powróćmy do rozważanego przykładu. Na rysunku 2 przedstawiony jest proces konstrukcji drzewa do momentu gdy powstanie $n = 10$ liści. Na rysunku 3 są przedstawione następne 3 iteracje, za każdym razem rozwijamy liść zaznaczony na szaro i usuwamy pewną zaznaczoną na szaro krawędź. Rozwijanie ostatniego zaznaczonego liścia nie zmniejsza kosztu i algorytm się zatrzymuje.

W algorytmie można pominąć konstrukcję drzewa i operować jedynie na wagach liści. Dane, na których działamy, to co najwyżej n liczb interpretowanych jako aktualne wagi liści.

Załóżmy, że mamy strukturę danych, która pozwala na wykonywanie następujących operacji:

- *ExtractMin*(**var** S : *struktura*) : **integer** — podaje najmniejszy element znajdujący się w strukturze i usuwa go,
- *Add*(**var** S : *struktura*; x : **integer**) — dodaje x do struktury, jeśli liczba elementów w strukturze jest większa niż n, to zostawia w niej tylko n najmniejszych elementów (a pozostałe wyrzuca),

- *Sum*(**var** *S* : *struktura*) : **integer** — zwraca sumę wszystkich elementów w strukturze,
- *Count*(**var** *S* : *struktura*) : **integer** — zwraca liczbę elementów w strukturze,
- *Init*(**var** *S* : *struktura*) — inicjuje strukturę tak, że zawiera jedynie element 0.

Dysponując opisaną strukturą, algorytm możemy zapisać następująco:

```
minSuma := +∞;
Init(S);
for i := 1 to k do Add(S, W(i));
while (Count(S) < n) or ((n = Count(S)) and (Sum(S) < minSuma))
                                                        do begin
  if (n = Count(S)) and (Sum(S) < minSuma) then
    minSuma := Sum(S);
  x := ExtractMin(S);
  for i := 1 to k do Add(S, x + W(i))
end
```

Strukturę można zaprogramować jako dwa **kopce** (opis kopca znajdzie czytelnik w [9]). Elementy jednego kopca są uporządkowane rosnąco, drugiego — malejąco. Każdy element struktury S jest wkładany do obu kopców, przy czym każda kopia posiada wskaźnik na swojego „bliźniaka” (który to wskaźnik trzeba aktualizować przy wszystkich operacjach na kopcu). Dla takiej implementacji struktury, złożoność operacji *ExtractMin* i *Add* wynosi $O(\log n)$. Ponieważ maksymalna liczba obrotów pętli wynosi n, cały algorytm ma złożoność $O(kn \log n)$.

TESTY

Do oceny rozwiązań zawodników wykorzystano 10 testów. Oto ich krótka charakterystyka:

- NAJ0.IN — test z treści zadania.
- NAJ1.IN, NAJ2.IN — proste testy poprawnościowe.
- NAJ3.IN, NAJ4.IN — test, dla którego wartości zmiennych przekraczają wartość typu **word**.
- NAJ5.IN–NAJ10.IN — testy wydajnościowe.

Andrzej Pelc
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Gra Ulama

Wybitny polski matematyk Stanisław Ulam zaproponował następującą grę dwuosobową.

Gracz A wybiera ze zbioru liczb całkowitych $X = \{0, 1, ..., 2047\}$ *liczbę* x*, nie zdradzając jej graczowi B.*

Gracz B musi dowiedzieć się, jaką liczbę wybrał gracz A. W tym celu może zadawać pytania postaci: „Czy x *jest elementem zbioru* Y*?”, gdzie* Y *jest dowolnym podzbiorem zbioru* X*.*

Gracz A udziela odpowiedzi „TAK” lub „NIE”, przy czym wolno mu co najwyżej raz skłamać.

Twoim zadaniem jest znalezienie i zaprogramowanie strategii zadawania pytań dla gracza B, która pozwoli mu wskazać poprawnie liczbę wybraną przez gracza A, przy możliwie najmniejszej liczbie zadanych pytań.

Przykład

Rozważmy grę Ulama dla zbioru $X = \{0, 1, 2, 3\}$*. Przyjrzyjmy się następującemu ciągowi pytań i odpowiedzi:*

B: Czy x *należy do* $\{0, 1\}$*?*
A: TAK
B: Czy x *należy do* $\{0\}$*?*
A: TAK

W tym miejscu B nie może jeszcze być pewnym, że liczbą wybraną przez A jest 0, ponieważ A mógł skłamać odpowiadając na drugie pytanie.

B: Czy x *należy do* $\{0\}$*?*
A: NIE

B nie wie, która z dwóch ostatnich odpowiedzi jest prawdziwa.

B: Czy x *należy do* $\{0\}$*?*
A: NIE

Szukaną liczbą na pewno nie jest 0. Odpowiedź na pierwsze pytanie musiała być prawdziwa, zatem szukaną liczbą jest 1.

B: x=1.

Zadanie

Twoim zadaniem jest zaprogramowanie strategii zadawania pytań umożliwiającej graczowi B znalezienie liczby x *przy pomocy jak najmniejszej liczby pytań. Rolę gracza A będzie pełnił program dostarczony przez organizatorów.*

Musisz zaprogramować moduł zawierający następujące trzy procedury (funkcje w przypadku języka C):

- `nowa_gra` — *ta procedura będzie wywoływana na początku każdej gry i tylko wtedy, możesz za jej pomocą dokonywać inicjalizacji swoich struktur danych;*
- `daj_pytanie` — *ta procedura służy do zadawania pytania i w tym celu powinna odpowiednio wypełniać tablicę pytanie opisaną w następnym paragrafie;*
- `analizuj_odpowiedz` — *ta procedura powinna odczytać z opisanej dalej zmiennej globalnej odpowiedź na ostatnio zadane pytanie i dokonać analizy tej odpowiedzi, jeśli w wyniku analizy ostatniej odpowiedzi liczba* x *zostanie zidentyfikowana, procedura powinna zasygnalizować ten fakt nadając właściwe wartości odpowiednim zmiennym globalnym.*

Program nadzorujący grę na początku wywoła napisaną przez Ciebie procedurę `nowa_gra` *po czym cyklicznie będzie wykonywał następujęce czynności:*

- *wywołanie procedury* `daj_pytanie`
- *odpowiadanie na pytanie*
- *wywołanie procedury* `analizuj_odpowiedz`

do momentu, gdy Twój program stwierdzi, że odgadł liczbę x (procedura `analizuj_odpowiedz` *umieści właściwą wartość w odpowiedniej zmiennej).*

UWAGA: Nie zakładaj, że program symulujący gracza A faktycznie ustala liczbę do odgadnięcia przed rozpoczęciem gry. Może dobierać ją w trakcie gry w taki sposób, aby zmusić Twój program do zadania największej liczby pytań lecz by po zakończeniu gry, kiedy znana jest „pomyślana" liczba oraz lista zadanych pytań i odpowiedzi, nie można było udowodnić, że gracz A skłamał więcej niż raz. Tak więc, powinieneś dążyć do tego, aby liczba pytań, jakie Twój program będzie musiał zadać w najgorszym przypadku, była jak najmniejsza.

Komunikacja

Komunikacja pomiędzy napisanym przez Ciebie modułem, a programem nadzorującym grę odbywa się przez zmienne globalne.

Zadanie pytania „Czy x należy do Y?" w procedurze `daj_pytanie` *polega na wypełnieniu tablicy* pytanie *typu* **array***[0..2047]* **of boolean** *(w języku C tablicy* **char** pytanie *[2048]) w taki sposób, że liczba i jest elementem Y wtedy i tylko wtedy, gdy* pytanie*[*i*]* = TAK*, gdzie* TAK *to predefiniowana stała o wartości* **true** *w Pascalu oraz 1 w C (podobnie predefiniowano stałą* NIE *o wartości* **false** *w Pascalu oraz 0 w C).*

Odpowiedź na zadane pytanie jest umieszczana w zmiennej odpowiedz *typu* **boolean** *(w języku C typu* **char***):* odpowiedz = TAK *wtedy i tylko wtedy, gdy odpowiedź jest twierdząca.*

O odnalezieniu szukanej liczby x Twój program powinien poinformować w procedurze `analizuj_odpowiedz`*, zapisując w zmiennej* wiem *typu* **boolean** *(w języku C typu* **char***)* TAK*, a następnie w zmiennej* x *znalezioną liczbę. Podanie odpowiedzi kończy grę, jeśli jest to odpowiedź błędna, gracz B przegrywa.*

Katalog i pliki

Programujący w Pascalu powinni przygotowywać swoje rozwiązanie w katalogu `C:\GRAPAS`*, natomiast piszący w* `C` *i* `C++` *w katalogu* `C:\GRAC`*.*

W katalogu `C:\GRAPAS` *(odpowiednio* `C:\GRAC`*) znajdziesz następujące pliki:*

- `gramod.pas` *(odpowiednio* `gramod.h` *i* `gramod.c`*) — moduł zawierający podstawowe definicje oraz deklaracje zmiennych komunikacyjnych;*
- `gra.pas` *— szkielet modułu grającego; powinieneś uzupełnić ten plik definicjami funkcji* `nowa_gra`*,* `daj_pytanie` *i* `analizuj_odpowiedz` *(dla piszących w języku C nagłówki powyższych 3 funkcji znajdują się w pliku* `gra.h`*, musisz napisać plik* `gra.c` *z definicjami tych funkcji);*
- `rywal.pas` *(odpowiednio* `rywal.c`*) — przykładowy program nadzorujący grę i korzystający z Twojego modułu, możesz z jego pomocą sprawdzić, czy Twój moduł spełnia specyfkację zadania.*

Wyjście

Wynikiem Twojej pracy powinien być zapisany na dyskietce tylko jeden plik `gra.pas` *lub* `gra.c`*, ale nie oba jednocześnie.*

Uwagi dla piszących w C i C++

Piszący w języku C (C++) spotkają się z koniecznością kompilacji programu wieloplikowego. Aby tego dokonać należy utworzyć projekt poleceniem „New Project" z menu „Project", po czym dodać (klawisz „Insert") do nowo utworzonego projektu wszystkie pliki `*.c`*. Od tego momentu polecenia „Make" i „Build" będą automatycznie kompilować i łączyć wszystkie pliki wymienione w projekcie.*

ROZWIĄZANIE

Każdy etap gry po odpowiedzi gracza $\mathcal{A}$ kodujemy jako parę liczb naturalnych (x, y) zwaną **stanem gry**. x jest mocą „zbioru prawd": zbioru tych liczb, które spełniają wszystkie dotychczasowe odpowiedzi. y

jest mocą „zbioru kłamstw": zbioru tych liczb, które spełniają wszystkie dotychczasowe odpowiedzi, poza jedną. Oczywiście tylko element sumy tych zbiorów może być prawidłowym rozwiązaniem, inne liczby wskazywałyby bowiem, że gracz $\mathcal{A}$ skłamał co najmniej 2 razy.

Intuicyjnie jest jasne, że graczowi $\mathcal{B}$ trudniej sobie poradzić z elementami zbioru prawdy, bo na temat tych elementów gracz $\mathcal{A}$ może jeszcze skłamać, podczas gdy, jeśli wybrana liczba jest w zbiorze kłamstw, musi on do końca mówić prawdę. Jeśli do końca gry pozostało jeszcze j pytań, nadajemy elementom zbioru prawd wagę $j+1$ odpowiadającą j możliwościom kłamstwa w każdej z pozostałych do końca odpowiedzi plus możliwości powstrzymania się od kłamstwa. Nadajmy też elementom zbioru kłamstwa wagę 1 odpowiadającą jedynej w ich przypadku możliwości: stałego mówienia prawdy. Łączna **waga** stanu gry (x,y) wynosi więc $w_j(x,y) = x(j+1)+y$.

Każde pytanie w stanie gry (x,y) można potraktować jako pytanie typu: *czy nieznana liczba jest w danym podzbiorze P wielkości p zbioru prawdy lub w danym podzbiorze K wielkości k zbioru kłamstw?* Takie pytanie można zakodować jako „(p,k)?" i odpowiedzi TAK/NIE prowadzą do stanów odpowiednio (x_1,y_1) i (x_2,y_2), gdzie $x_1 = p$, $x_2 = x-p$, $y_1 = k+x-p$, $y_2 = y-k+p$. Łatwo zauważyć, że prawdziwa jest w tym wypadku równość:

$$w_j(x,y) = w_{j-1}(x_1,y_1) + w_{j-1}(x_2,y_2).$$

Równość ta sugeruje optymalną strategię dla gracza $\mathcal{B}$: należy tak zadawać pytania, by dzielić wagę stanu na połowy. Przy tej strategii gracz $\mathcal{B}$ zawsze znajduje nieznaną liczbę w co najwyżej 15-tu pytaniach i można pokazać, że jest to minimalna liczba pytań w najgorszym przypadku, bowiem $w_{15}(2^{11},0) = 2^{15}$.

A oto strategia. Stan początkowy to $(2^{11},0)$. Zadajemy pytanie $(2^{10},0)$?. Obie odpowiedzi na to pytanie prowadzą do stanów $(2^{10},2^{10})$. Następnie zadajemy pytanie $(2^9,2^9)$?, uzyskując przy obu odpowiedziach stany $(2^8,2^9)$, itd. Przez pierwszych 11 pytań w stanie (x,y) zadajemy pytanie $(x/2,y/2)$. Po tych 11 pytaniach, niezależnie od odpowiedzi uzyskujemy stan $(1,11)$. Pozostały nam 4 pytania, zatem $w_4(1,11) = 16$. Zadajemy pytanie $(1,4)$?. Odpowiedzi TAK/NIE prowadzą odpowiednio do stanów $(1,4)$ i $(0,8)$. Drugi z nich jest łatwy: zwykłe wyszukiwanie binarne przy pomocy trzech pytań. W pierwszym zaś zadajemy pytanie $(1,1)$? prowadzące do stanów $(1,1)$ i $(0,4)$. Zostały dwa pytania. Znów stan $(0,4)$ jest łatwy, a w pierwszym zadajemy pytanie $(1,0)$? uzyskując możliwe stany $(1,0)$ i $(0,2)$. W pierwszym przypadku poszukiwanie jest zakończone, a w drugim jedno dodatkowe pytanie wystarcza.

Przy implementowaniu tej strategii należy pamiętać parę zbiorów (X,Y), gdzie X jest aktualnym zbiorem prawd, a Y aktualnym zbiorem kłamstw. Pytanie (p,k)? należy interpretować jako pytanie *Czy nieznana liczba jest w zbiorze T?*, gdzie T jest sumą zbiorów pierwszych p liczb ze zbioru X i pierwszych k liczb ze zbioru Y.

TESTY

Testy wykonano przy pomocy modułu arbitra, którego kod był konsolidowany z kodem programów dostarczonych przez zawodników. Program arbitra był sparametryzowany tak, by mógł grać wg różnych strategii, poczynając od najprostszej, polegającej na wybraniu liczby i odpowiadaniu na pytania w odniesieniu do tej liczby, a kończąc na strategii gracza $\mathcal{A}$ wymuszającej zadanie 15 pytań. Szczegóły znajdzie Czytelnik w załączonym na dyskietce kodzie źródłowym.

Krzysztof Sobusiak
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Łamane płaskie

Rozważmy kartezjański układ współrzędnych narysowany na kartce papieru. Łamaną płaską w tym układzie nazywamy łamaną, którą można wykreślić prowadząc ołówek bez odrywania od papieru w kierunku od lewej do prawej strony kartki i taką, że każda prosta zawierająca odcinek łamanej jest nachylona do osi OX pod kątem należącym do przedziału domkniętego $[-45°, 45°]$.

Na kartce narysowano n różnych punktów o obu współrzędnych całkowitych. Jaka jest najmniejsza liczba łamanych płaskich, które należałoby wykreślić, żeby każdy z n punktów należał do co najmniej jednej z nich?

Przykład

Dla 6 punktów o współrzędnych $(1, 6)$, $(10, 8)$, $(1, 5)$, $(2, 20)$, $(4, 4)$, $(6, 2)$ najmniejszą liczbą łamanych płaskich pokrywających te punkty jest 3.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego LAM.IN liczbę punktów i ich współrzędne;*
- *oblicza najmniejszą liczbę łamanych płaskich, które należałoby wykreślić, żeby pokryły one wszystkie punkty;*
- *zapisuje wynik w pliku tekstowym LAM.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego LAM.IN znajduje się dodatnia liczba całkowita n nie większa od 30000. Jest to liczba punktów.

W kolejnych n wierszach znajdują się współrzędne punktów.

Każdy wiersz zawiera dwie liczby całkowite x, y oddzielone pojedynczym odstępem, $0 \leqslant x \leqslant 30000$, $0 \leqslant y \leqslant 30000$. Liczby w wierszu $i+1$, $1 \leqslant i \leqslant n$, pliku LAM.IN są współrzędnymi i-tego punktu.

Wyjście

Twój program powinien zapisać jedną liczbę całkowitą w pierwszym i jedynym wierszu pliku wyjściowego LAM.OUT. Tą liczbą powinna być najmniejsza liczba łamanych płaskich, które należy wykreślić, żeby pokryć wszystkie dane punkty.

Przykład

Dla pliku wejściowego LAM.IN:

```
6
1 6
10 8
1 5
2 20
4 4
6 2
```

poprawnym rozwiązaniem jest plik wyjściowy LAM.OUT:

```
3
```

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

ROZWIĄZANIE

Kluczową dla rozwiązania zadania „operacją” jest obrót kartki papieru o 45 stopni w lewo, tzn. obrót każdego punktu o 45 stopni w lewo względem początku układu współrzędnych. Wtedy łamaną płaską będzie łamana wykreślona bez odrywania ołówka od papieru w kierunku od lewej do prawej i taka, że każda prosta zawierająca odcinek łamanej jest nachylona do osi OX pod kątem należącym do przedziału domkniętego $[0°, 90°]$.

Aby rozwiązać takie zmodyfikowane zadanie zastosujemy technikę zwaną **zamiataniem**, często spotykaną w algorytmach geometrycznych.

Sortujemy najpierw wszystkie punkty niemalejąco względem współrzędnych x, przy czym jeśli dwa punkty mają równe współrzędne x wtedy przyjmujemy, że wcześniejszy w porządku jest punkt o mniejszej współrzędnej y (posortować możemy algorytmem Heapsort lub innym algorytmem sortującym o złożoności $O(n \log n)$, zob. [12]).

Wysokością łamanej płaskiej będziemy nazywać wspólrzędną y ostatniego jej punktu. **i-tym prefiksem** zbioru łamanych płaskich nazwiemy początkowe fragmenty łamanych z tego zbioru, rozpięte na i pierwszych punktach (w określonym wyżej porządku). **i-tym przekrojem** rozwiązania nazwiemy malejący ciąg $a_1, \ldots, a_{p_i}$ wysokości łamanych w i-tym prefiksie zbioru łamanych będących rozwiązaniem zadania.

Nasz algorytm będzie przeglądał uporządkowane punkty tworząc kolejne przekroje rozwiązania optymalnego.

Przekrój zerowy jest oczywiście pusty.

Dla kolejnego punktu P_i o współrzędnych (x_i, y_i) znajdujemy w $(i-1)$-tym przekroju maksymalny element a_k spełniający nierówność $a_k \leqslant y_i$. Punkt P_i podczepimy do łamanej o tej właśnie wysokości — stanie się on jej ostatnim punktem, zatem i-ty przekrój powstanie z przekroju $(i-1)$-ego poprzez zamianę a_k na y_i.

Jeśli wszystkie łamane mają większe wysokości niż y_i, wówczas tworzymy nową łamaną złożoną tylko z tego jednego punktu, czyli przekrój i-ty otrzymujemy z przekroju $(i-1)$-ego poprzez dodanie elementu y_i na koniec ciągu.

Rozwiązaniem zadania jest długość n-tego przekroju.

Zauważmy, że kolejne przekroje możemy obliczać w jednej tablicy. Wyszukiwań łamanych do rozszerzenia możemy wtedy dokonywać binarnie w czasie $O(\log n)$. Takich wyszukiwań wykonamy dokładnie n, zatem łącznie ze wstępną fazą sortowania złożoność naszego algorytmu wynosi $O(n \log n)$.

Opisany powyżej algorytm należy do klasy algorytmów **zachłannych**, tzn. takich, które znajdują optymalne rozwiązanie dla danego problemu wykonując lokalne działania, które są w danej chwili najkorzystniejsze. Takie „intuicyjnie najlepsze” zachowanie algorytmu nie musi jednak zawsze prowadzić do rozwiązania optymalnego. Na szczęście opisany wyżej algorytm zawsze znajduje optymalne rozwiązanie, co udowodniamy poniżej.

Powiemy, że przekrój $a_1, \ldots, a_p$ jest **niższy** od przekroju $b_1, \ldots, b_r$, jeśli $p \leqslant r$ oraz istnieje podciąg $b'_1, \ldots, b'_p$ ciągu $b_1, \ldots, b_r$ taki, że dla każdego k zachodzi $a_k \leqslant b'_k$, tzn. dla każdej łamanej z przekroju $a_1, \ldots, a_p$ mamy łamaną o nie mniejszej wysokości w przekroju $b_1, \ldots, b_r$.

Pokażemy indukcyjnie, że dla dowolnego rozwiązania B (niekoniecznie optymalnego), i-ty przekrój rozwiązania A wyznaczanego przez nasz algorytm jest niższy od i-tego przekroju rozwiązania B.

Dla $i = 0$ powyższe stwierdzenie jest w oczywisty sposób prawdziwe.

Załóżmy więc, że $(i-1)$-szy przekrój $a_1, \ldots, a_p$ rozwiązania A jest niższy od $(i-1)$-ego przekroju $b_1, \ldots, b_r$ rozwiązania B. Teraz musimy rozpatrzeć 2 przypadki:

- punkt $P_i(x_i, y_i)$ jest podłączony w rozwiązaniu B do łamanej o wysokości b_k.
- punkt $P_i(x_i, y_i)$ w rozwiązaniu B jest początkiem nowej łamanej.

W obu z nich można w miarę łatwo pokazać, że i-ty przekrój rozwiązania A jest niższy od i-tego przekroju rozwiązania B. Pozostawiamy dowód tego faktu Czytelnikowi jako pouczające ćwiczenie.

Jak obliczyć współrzędne punktu (x', y') powstałego przez obrót punktu (x, y) o 45 stopni w lewo względem początku układu współrzędnych?

W dowolnej książce poświęconej geometrii analitycznej znajdujemy (lub lepiej: sami wyprowadzamy) wzory dla dowolnego kąta ϕ:

$$x' = x\cos\phi - y\sin\phi,$$
$$y' = x\sin\phi + y\sin\phi.$$

Ponieważ $\phi = 45°$, to:

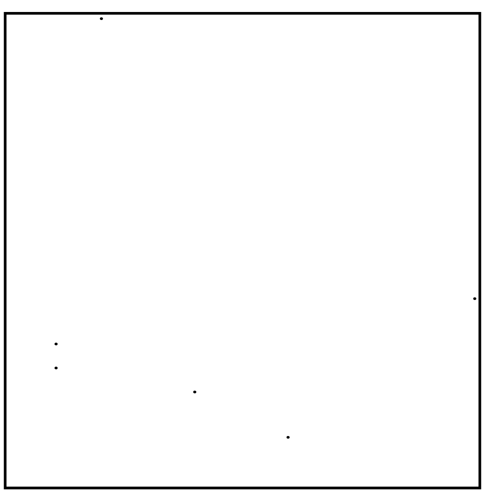
$$x' = \frac{\sqrt{2}}{2}(x - y),$$
$$y' = \frac{\sqrt{2}}{2}(x + y).$$

Zwykle we wszelkich algorytmach staramy się unikać (jeśli to możliwe) liczb rzeczywistych ze względu na ich niedokładną reprezentację. Z tego powodu skalujemy wszystkie punkty, dzieląc ich obie współrzędne przez $\sqrt{2}/2$. Ostatecznie mamy:

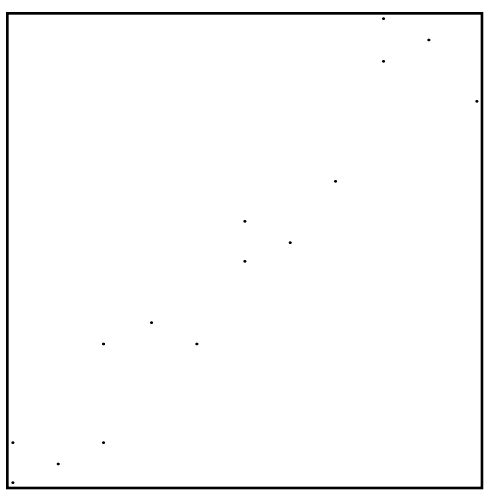
$$x' = x - y,$$
$$y' = x + y.$$

TESTY

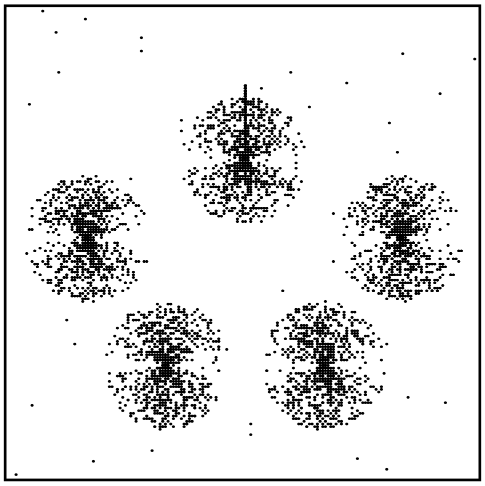
Do sprawdzenia rozwiązań zawodników użyto 11-tu testów LAM0.IN–LAM10.IN. Test LAM0.IN był testem z treści zadania. Test LAM1.IN był prostym testem poprawnościowym, test LAM2.IN — prostym testem wydajnościowym, test LAM3.IN — testem losowym. Punkty w teście LAM4.IN tworzyły napis „Olimpiada Informatyczna". Pozostałe testy były trudnymi testami wydajnościowymi. Poniżej przedstawiamy schematy ułożenia punktów w poszczególnych testach.



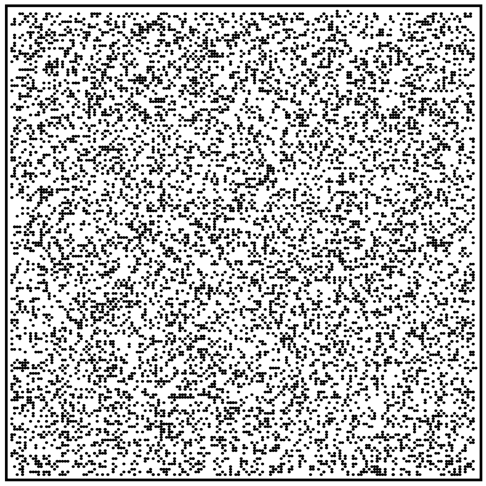
LAM0.IN $n = 6$



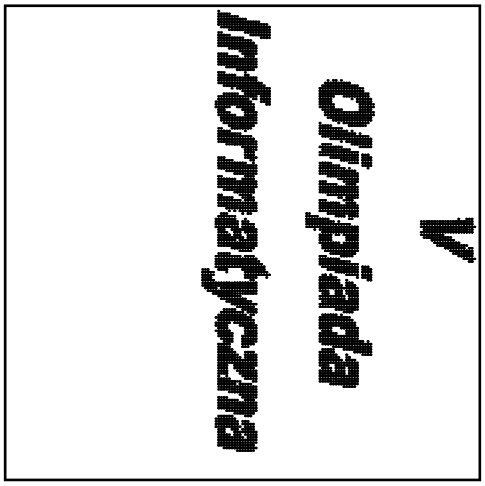
LAM1.IN $n = 15$



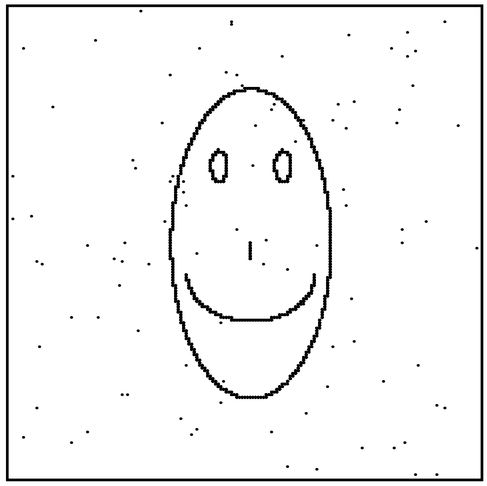
LAM2.IN $n = 5000$



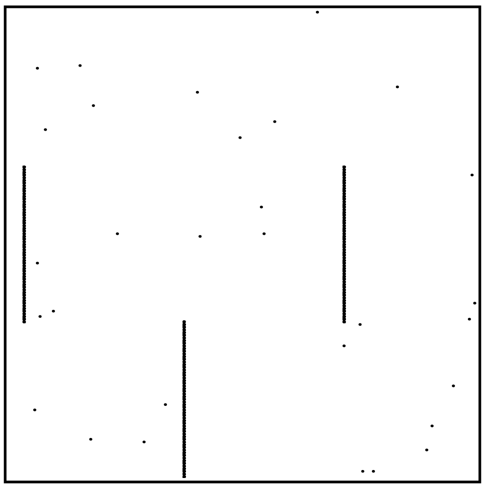
LAM3.IN $n = 10000$



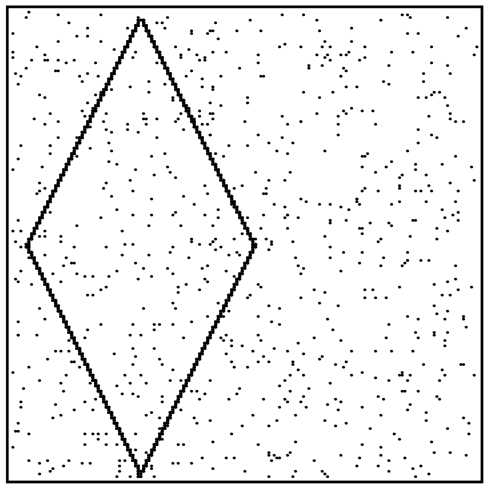
LAM4.IN $n = 15000$



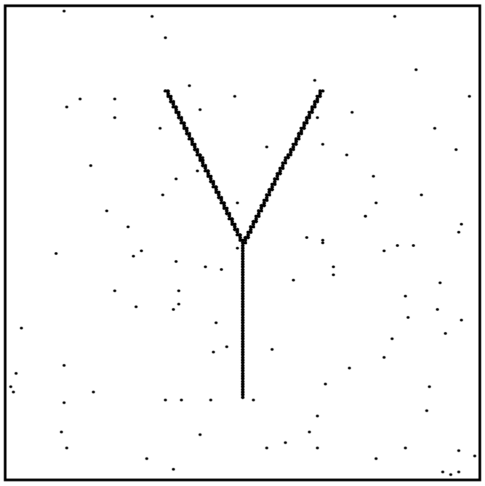
LAM5.IN $n = 17000$



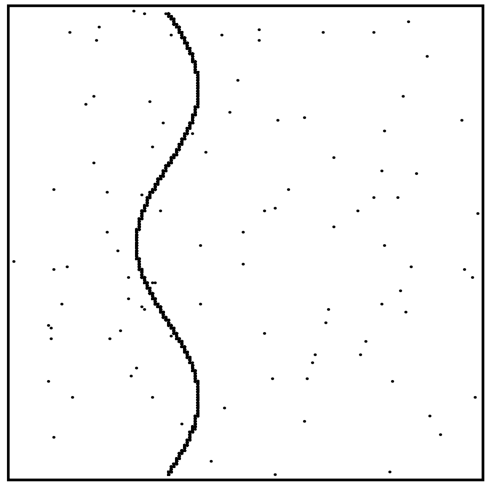
LAM6.IN $n = 30000$



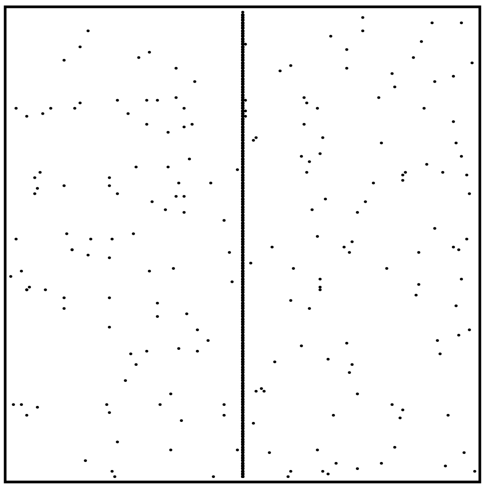
LAM7.IN $n = 30000$



LAM8.IN $n = 30000$



LAM9.IN $n = 30000$



LAM10.IN $n = 30000$

Prostokąty

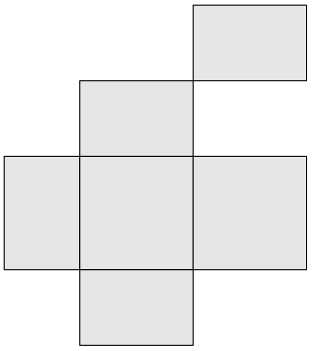
Na płaszczyźnie narysowano n prostokątów, których boki są równoległe do osi współrzędnych i wierzchołki mają obie współrzędne całkowite.

Przyjmujemy że:

- *każdy prostokąt jest blokiem,*
- *jeśli dwa różne bloki mają wspólny odcinek to tworzą one nowy blok, w przeciwnym przypadku mówimy, że są rozłączne.*

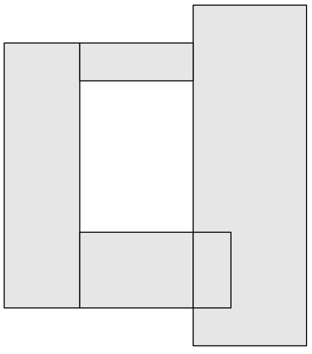
Przykłady

Prostokąty na rysunku a tworzą dwa rozłączne bloki.



Rys. a

Prostokąty na rysunku b tworzą jeden blok.



Rys. b

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego PRO.IN liczbę prostokątów oraz współrzędne ich wierzchołków;*
- *znajduje liczbę rozłącznych bloków utworzonych przez te prostokąty;*
- *zapisuje wynik w pliku tekstowym PRO.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego PRO.IN znajduje się liczba całkowita n, $1 \leqslant n \leqslant 7000$. Jest to liczba prostokątów.

W następnych n wierszach są zapisane współrzędne wierzchołków prostokątów. Każdy prostokąt jest opisany za pomocą czterech liczb: współrzędnych x i y lewego dolnego wierzchołka oraz współrzędnych x i y prawego górnego wierzchołka. Są to liczby całkowite nieujemne nie większe niż 10000.

Wyjście

W pierwszym i jedynym wierszu pliku PRO.OUT należy zapisać jedną liczbę całkowitą: liczbę rozłącznych bloków utworzonych przez dane prostokąty.

Przykład

Dla pliku tekstowego PRO.IN:

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

poprawnym rozwiązaniem jest plik wyjściowy PRO.OUT:

```
2
```

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

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

Każdy prostokąt ma dodatnie pole.

ROZWIĄZANIE

Zadanie można rozwiązać na kilka sposobów. Na początku omówimy rozwiązania polegające na przeszukiwaniu grafów. Zacznijmy od opisania grafu. Wierzchołkami będą podane prostokąty. Powiemy, że dwa prostokąty są połączone krawędzią, jeśli mają wspólny odcinek (tzn. gdy tworzą blok). Każdy graf rozpada się na tzw. ***składowe***. Składową grafu nazywamy taki jego maksymalny podgraf, w którym każde dwa wierzchołki można połączyć ścieżką. Zauważmy, że dwa prostokąty wchodzą w skład tego samego bloku, jeśli w tak skonstruowanym grafie istnieje między nimi ścieżka. Zatem szukana liczba bloków jest równa liczbie składowych skonstruowanego przez nas grafu. Tę liczbę składowych można wyznaczyć za pomocą procedur przeszukiwania grafów.

Jedną z metod przeszukiwania grafów jest metoda przeszukiwania grafów w głąb (ang. *Depth First Search*, w skrócie *DFS*). Polega ona na tym, że najpierw zaznaczamy w specjalnej tablicy, że żadnego wierzchołka dotychczas nie odwiedziliśmy, a następnie rozpoczynamy przeszukiwanie. Wybieramy wierzchołek, którego dotychczas nie odwiedziliśmy, zaznaczamy, że został już odwiedzony i wyruszamy z niego do któregokolwiek jego nie odwiedzonego sąsiada. Zaznaczamy, że tego sąsiada odwiedziliśmy i podążamy dalej. Jeśli dojdziemy do wierzchołka, którego wszystkich sąsiadów już odwiedziliśmy, musimy się cofnąć i poszukać nowego wierzchołka, którego nie odwiedziliśmy dotychczas. Robimy to tak długo, aż odwiedzimy wszystkie wierzchołki dostępne z wierzchołka początkowego. Wtedy zapisujemy „na koncie" jedną składową i szukamy wierzchołka, którego nie odwiedziliśmy. Zaznaczamy, że został odwiedzony i ponawiamy przeszukiwanie. Po jego zakończeniu zaznaczamy „na koncie" drugą składową i powtarzamy to postępowanie dotąd, aż odwiedzimy wszystkie wierzchołki. Najprościej można zaimplementować ten algorytm za pomocą następujacej procedury rekurencyjnej:

```
procedure DFS (v : Wierzcholek);
  var
    w : Wierzchołek;
  begin
    Odwiedzony[v] := true;
    for w in Sasiedzi[v] do
      if not Odwiedzony[w] then DFS(w)
  end;
procedure SzukajDFS (var liczba : integer);
  var
    v : Wierzcholek;
  begin
    forall v do Odwiedzony[v] := false;
    liczba := 0;
    forall v do
      if not Odwiedzony[v] then begin
        DFS(v);
        liczba := liczba + 1
      end
  end;
```

Można również przeszukiwać graf wszerz (ang. *Breadth First Search*, w skrócie *BFS*). Przede wszystkim tworzymy kolejkę — listę, do której dopisujemy kolejne elementy z jednej strony i pobieramy z drugiej, tak jak to się dzieje w kolejce do sklepu (Starsi jeszcze pamiętają to zjawisko!). Do kolejki pustej dopisujemy pierwszy wierzchołek. Teraz zaczynamy przeszukiwanie. Robimy to w ten sposób, że wybieramy pierwszy wierzchołek, zaznaczamy, że został już odwiedzony i dołączamy do kolejki tych jego sąsiadów, którzy nie zostali jeszcze odwiedzeni. Potem pobieramy pierwszy wierzchołek z kolejki i dołączamy do kolejki wszystkich nieodwiedzonych sąsiadów tego wierzchołka itd. W chwili, gdy kolejka stanie się pusta, odwiedzone zostaną wszystkie wierzchołki znajdujące się w tej samej składowej co wierzchołek początkowy. Zapisujemy „na koncie" jedną składową, wybieramy dowolny nieodwiedzony wierzchołek i powtarzamy procedurę. Można ten sposób przeszukiwania zapisać jak następuje:

```
procedure BFS(v : Wierzcholek);
  var
    K : Kolejka;
    u, w : Wierzcholek;
  begin
    K := PUSTA;
    Odwiedzony[v] := true;
    Dopisz(v, K);
    while K ≠ PUSTA do begin
      u := Pobierz(K);
      for w in Sasiedzi[u] do
        if not Odwiedzony[w] then begin
          Odwiedzony[w] := true;
          Dopisz(w, K)
        end
```

```
      end
    end;
procedure SzukajBFS(var liczba : integer);
  var
    v : Wierzcholek;
  begin
  for v do Odwiedzony[v] := false;
    liczba := 0;
    forall v do
      if not Odwiedzony[v] then begin
        BFS(v);
        liczba := liczba + 1
      end
  end;
```

Oczywiście procedura *Dopisz*(v, K) dopisuje wierzchołek v na końcu kolejki K, a funkcja *Pobierz*(K) zwraca pierwszy wierzchołek kolejki i usuwa go z niej. Zauważmy, że przedstawiona wyżej procedura *SzukajBFS* niewiele różni się od procedury *SzukajDFS*.

W obu metodach przeszukiwania problemem jest zbudowanie grafu w pamięci. Przy maksymalnej liczbie 7000 prostokątów liczba krawędzi może sięgać nawet 49 milionów! Będzie tak, na przykład wtedy, gdy wszystkie prostokąty będą identyczne. Oczywiście tak dużego grafu nie da się zbudować w pamięci operacyjnej i program próbujący to zrobić zakończy się błędem. Budowanie grafu w pamięci operacyjnej nie jest jednak konieczne. Sąsiadów danego wierzchołka szukamy przecież na bieżąco, przeszukując wszystkie wierzchołki. Zauważmy, że nigdzie nie wykorzystujemy krawędzi grafu w inny sposób, niż do stwierdzenia, czy rozważany wierzchołek jest sąsiadem wierzchołka, którym aktualnie się zajmujemy. Taki sposób implementacji wymaga już znacznie mniej pamięci, choć jego czas działania pozostanie taki sam. Ten sposób implementacji, w którym budujemy graf w pamięci operacyjnej, wymaga czasu rzędu n^2 i takiego samego rzędu pamięci, drugi, w którym graf budujemy „na bieżąco", takiego samego czasu, ale pamięci tylko rzędu n. Jest to znaczna poprawa; taki program będzie mógł pomyślnie zakończyć działanie nawet dla największych spodziewanych danych.

Następna metoda rozwiązania zadania polega na wykorzystaniu nowej struktury danych, tzw. zbiorów rozłącznych. Na początku wczytujemy do pamięci dane wszystkich prostokątów. Następnie tworzymy tablicę S typu **array**[1..*MaxLiczbaProstokatow*] **of integer** i całą ją wypełniamy liczbą -1. Wartość $S[i] = -1$ oznacza, że i-ty prostokąt stanowi samodzielny blok. Na początku przyjmujemy, że tak jest. Potem zbadamy wszystkie możliwe połączenia bloków. Przypuśćmy więc, że znajdziemy pierwsze połączenie bloków: niech prostokąt drugi stanowi jeden blok z prostokątem piątym. Wtedy zmienimy jedną wartość: albo położymy $S[2]$:=5, albo $S[5]$:=2. Chwilowo jest to obojętne. W każdym przypadku tylko jeden prostokąt ma przypisaną liczbę -1, drugi zaś ma przypisany **numer** tego pierwszego prostokąta. Ten prostokąt, który ma przypisaną liczbę -1, nazwiemy **głową** bloku. Przyjmijmy dla ustalenia uwagi, że głową tego bloku jest prostokąt o numerze 2. Wtedy $S[2] = -1$ oraz $S[5] = 2$. Niech następnie okaże się, że prostokąty piąty i czwarty tworzą jeden blok (na razie nie zajmujemy się kolejnością, w jakiej wykrywamy te połączenia mniejszych bloków w większe; do tej kwestii jeszcze powrócimy). Wtedy szukamy głów obu bloków, w których są te dwa prostokąty. Dla prostokąta czwartego jest to łatwe: równość $S[4] = -1$ świadczy o tym, że jest on głową bloku, którego jest częścią. Dla prostokąta piątego jest trudniej: najpierw stwierdzamy, że $S[5] = 2$, co znaczy, iż prostokąt piąty jest w tym samym bloku co prostokąt drugi. Teraz równość $S[2] = -1$ pokazuje, że głową tego bloku jest prostokąt drugi. Wreszcie łączymy te dwa bloki. Możemy znów uczynić to na dwa sposoby: albo przypiszemy wartość $S[2] := 4$, czyniąc czwarty prostokąt głową tego bloku, albo przypiszemy $S[4] := 2$, dołączając czwarty prostokąt do bloku, którego głową jest nadal prostokąt drugi. Oba rozwiązania wydają się być równoważne.

Po przebadaniu wszystkich możliwych połączeń bloków, tzn. po sprawdzeniu, czy i-ty prostokąt graniczy z j-tym prostokątem dla każdej pary różnych liczb $\{i, j\}$, w tablicy S będziemy mieli tyle liczb -1, ile jest bloków.

Zastanówmy się teraz, jakie czynności będą wykonywane w czasie uaktualniania tablicy S. Jedną z tych czynności będzie operacja *Find*(i), która prostokątowi o numerze i przypisze numer głowy bloku, którego częścią jest i-ty prostokąt. Druga czynność *Union*(i,j) polega na łączeniu w jeden dwóch bloków, których głowy mają numery i oraz j. Ta druga czynność jest łatwa: możemy albo przypisać $S[i]{:=}j$, albo $S[j]{:=}i$. Pierwsza jest realizowana za pomocą procedury:

```
function Find(i : integer) : integer;
  var
    j : integer;
  begin
    j := S[i];
    while j > 0 do begin
      i := j;
      j := S[i]
    end;
    Find := i
  end;
```

Nadszedł moment, w którym warto się zastanowić nad czasem wykonania poszczególnych czynności. Procedura *Union* jest wykonywana w stałym czasie. Procedura *Find* jest wykonywana w czasie proporcjonalnym do długości drogi (w tablicy S) przebywanej od i do głowy bloku. Okazuje się, że ta droga nie będzie wydłużała się zbytnio podczas wykonywania procedury *Union*, jeśli będziemy zawsze „podczepiać" mniejszy blok do głowy większego, przy czym przez mniejszy rozumiemy ten, w którym najdłuższa droga do głowy jest krótsza niż w drugim bloku. Niech zatem ujemna wartość $S[i]$ oznacza liczbę przeciwną do długości najdłuższej drogi do głowy w bloku, którego głową jest i-ty prostokąt. Na początku więc oczywiście $S[i] = -1$ dla wszystkich i. Potem procedura *Union* przybierze postać:

```
procedure Union(i, j : integer);
  begin
    if S[j] < S[i] then
      S[i] := j
    else begin
      if S[j] = S[i] then
        {wydłużamy drogę o 1}
        S[i] := S[i] - 1;
      S[j] := i
  end;
```

Wreszcie powiedzmy słowo o kolejności sprawdzania. Ponieważ trzeba sprawdzić wszystkie pary, więc możemy to zrobić za pomocą podwójnej pętli:

```
for i:=1 to LiczbaProstokatow do begin
  S[i] := -1;
  for j:=1 to i - 1 do
    Union(Find(i), Find(j))
end
```

Po wykonaniu tej pętli wystarczy zliczyć wszystkie głowy, tzn. wystarczy policzyć ile liczb ujemnych występuje w tablicy S. Ten sposób implementacji jest dość szybki (jest rzędu $n^2 \log n$) i pomimo, że wydaje się być wolniejszy od poprzednich (o czynnik $\log n$), daje szybko działające programy — między innymi dzięki swej prostocie. Warto jeszcze wspomnieć o tym, że podczas wykonywania procedury *Find* możemy zmodyfikować wszystkie wartości tablicy S na drodze od i do głowy, przypisując każdemu podczepionemu prostokątowi numer głowy bloku. To bardzo przyspieszy następne wyszukiwania i w praktyce da program nawet nieco szybszy od programów wykorzystujących metody teorii grafów.

INNE ROZWIĄZANIA

Warto tu zauważyć, że istnieje jeszcze inne rozwiązanie zadania, działające w czasie bliskim $O(n \log n)$. Nie będziemy go jednak opisywać ze względu na duży stopień zaawansowania wykorzystanych w nim technik i struktur danych. Z tych samych przyczyn tego rozwiązania nie brano pod uwagę przy ustalaniu czasów odcięcia dla testów.

TESTY

Oprócz przykładu z treści zadania, zastosowano 10 testów. Test PRO1.IN miał na celu sprawdzenie poprawności w sytuacji, w której pewne prostokąty stykały się tylko rogami lub fragmentami brzegów. Test PRO2.IN to „szachownica" wymiaru 10×10. Testy PRO3.IN–PRO6.IN obejmowały stosunkowo nieduże układy różnie rozmieszczonych prostokątów. Testy PRO7.IN–PRO10.IN obejmowały układy duże, po ok. 5000 prostokątów.