Szymon Acedański
Treść zadania, Opracowanie

Marek Cygan
Opracowanie, Program

OI, Etap III, dzień 1, 6–04–2005

Akcja komandosów

Na Pustyni Błędowskiej odbywa się w tym roku Bardzo Interesująca i Widowiskowa Akcja Komandosów (BIWAK). Podstawowym elementem BIWAK-u ma być neutralizacja bomby, która znajduje się gdzieś na pustyni, jednak nie wiadomo dokładnie gdzie.

Pierwsza część akcji to desant z powietrza. Z helikoptera krążącego nad pustynią, wyskakują pojedynczo, w ustalonej kolejności komandosi. Gdy któryś z komandosów wyląduje w jakimś miejscu, okopuje się i już z tego miejsca się nie rusza. Dopiero potem może wyskoczyć kolejny komandos.

Dla każdego komandosa określona jest pewna **odległość rażenia**. *Jeśli komandos przebywa w tej odległości (lub mniejszej) od bomby, to w przypadku jej ewentualnej eksplozji zginie. Dowództwo chce zminimalizować liczbę komandosów biorących udział w akcji, ale chce mieć też pewność, że w przypadku wybuchu bomby, przynajmniej jeden z komandosów przeżyje.*

Na potrzeby zadania przyjmujemy, że Pustynia Błędowska jest płaszczyzną, a komandosów, którzy się okopali, utożsamiamy z punktami. Mamy dany ciąg kolejno mogących wyskoczyć komandosów. Żaden z nich nie może opuścić swojej kolejki, tzn. jeśli i-ty komandos wyskakuje z samolotu, to wszyscy poprzedni wyskoczyli już wcześniej. Dla każdego z komandosów znamy jego odległość rażenia oraz współrzędne punktu, w którym wyląduje, o ile w ogóle wyskoczy.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy komandosów,*
- *wyznaczy minimalną liczbę komandosów, którzy muszą wyskoczyć,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita n ($2 \leqslant n \leqslant 2\,000$) — liczba komandosów. W kolejnych n wierszach opisani są komandosi — po jednym w wierszu. Opis każdego komandosa składa się z trzech liczb całkowitych: x, y i r ($-1\,000 \leqslant x, y \leqslant 1\,000$, $1 \leqslant r \leqslant 5\,000$). Punkt (x, y) to miejsce, gdzie wyląduje komandos, a r to jego odległość rażenia. Jeśli komandos znajdzie się w odległości r lub mniejszej od bomby, to w przypadku jej wybuchu zginie.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien zapisać jedną liczbę całkowitą — minimalną liczbę komandosów, którzy muszą wyskoczyć, aby zapewnić,

że co najmniej jeden z nich przeżyje. Jeśli nawet po zrzuceniu wszystkich komandosów, nie można mieć pewności, że któryś z nich przeżyje, to należy wypisać słowo NIE.

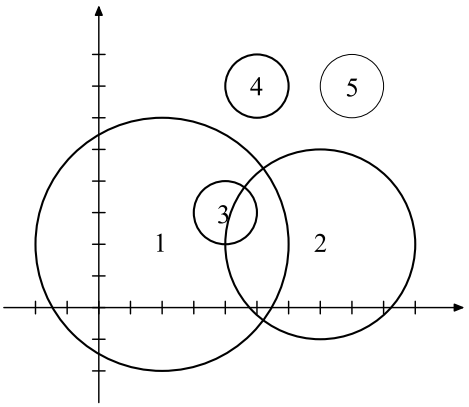
Przykład

Dla danych wejściowych:

```
5
2 2 4
7 2 3
4 3 1
5 7 1
8 7 1
```

poprawnym wynikiem jest:

```
4
```



Uwaga

To zadanie można rozwiązać używając typów zmiennopozycyjnych:

- *w Pascalu:* `double` *lub* `extended`,
- *w C oraz C++:* `double` *lub* `long double`.

Użycie typów `float` *lub* `real` *może spowodować błędy w obliczeniach związane z niedokładnością operacji zmiennopozycyjnych.*

Rozwiązanie

Analiza problemu

Pole rażenia każdego komandosa to koło. Bomba umieszczona w punkcie X zabije tych komandosów, których pola rażenia zawierają punkt X. Jeśli więc część wspólna pól rażenia zrzuconych komandosów jest pusta, to dowódcy mogą być pewni, że żadna bomba nie zabije wszystkich żołnierzy.

Ograniczenia na wielkość danych wejściowych podane w zadaniu mogą sugerować, iż złożoność czasowa oczekiwanych rozwiązań powinna być rzędu $O(n^2)$. Taki limit wydaje się być dość łatwy do osiągnięcia — można wymyślić sporo rozwiązań działających w takim lub podobnym czasie. Rozwiązanie wzorcowe zostało oparte na jednym z takich pomysłów zasługującym na szczególną uwagę z racji prostoty i łatwości w implementacji.

Rozwiązanie wzorcowe

Podstawowy problem, który musimy rozwiązać, to rozpoznanie dla zadanego zbioru kół $\mathcal{K} = \{K_1, K_2, \ldots, K_i\}$, czy część wspólna tych kół

$$S \quad = \quad K_1 \cap K_2 \cap \ldots \cap K_i$$

jest pusta. Oznaczmy przez X skrajnie prawy (o największej współrzędnej x) punkt zbioru S, tzn. zdefiniujmy

$$X \in S \text{ oraz dla każdego } Z \in S \text{ zachodzi } Z.x \leqslant X.x,$$

gdzie przez $V.x$ oznaczamy współrzędną x punktu V.

Obserwacja 1 *Punkt X ma następujące własności:*

(a) *jest wyznaczony jednoznacznie;*

(b) *leży na brzegu pewnego koła $K \in \mathcal{K}$.*

(c) *dla $i > 1$ jest skrajnie prawym punktem części wspólnej dwóch różnych kół $K, K' \in \mathcal{K}$.*

Dowód Własność (a) wynika z faktu, iż zbiór S jest wypukły i jego brzeg nie zawiera prostych fragmentów. Własność (b) jest także oczywista, bo gdyby nie zachodziła, to znaleźlibyśmy w zbiorze S punkt na prawo od X.

Własność (c) również można łatwo uzasadnić. Niech K będzie kołem, na którego brzegu leży punkt X. Ponieważ X jest skrajnie prawym punktem zbioru S, więc nie możemy przesunąć się na prawo idąc po obwodzie koła K i nie wychodząc poza zbiór S. To oznacza, iż:

- jesteśmy w skrajnie prawym punkcie koła K (jesteśmy więc w skrajnie prawym punkcie przecięcia koła K z dowolnym innym kołem ze zbioru $\mathcal{K}$);
- jesteśmy w punkcie przecięcia koła K z innym kołem K'.

Tak więc w obu przypadkach mamy własność (c). ■

Zdefiniujmy zbiór

$$P \quad = \quad \{Y : Y \text{ jest skrajnie prawym punktem przecięcia dwóch różnych kół } K, K' \in \mathcal{K}\}$$

i sformułujmy jego ważną własność, która stanie się podstawą algorytmu rozwiązania zadania.

Lemat 2 *Jeśli zbiór S jest niepusty, to punkt X należy do zbioru P i jest jego skrajnie lewym punktem.*

Dowód Niech $S \neq \emptyset$. Z poprzedniej obserwacji (punkt (c)) wiemy, że $X \in P$. Niech teraz $K, K' \in \mathcal{K}$ będą takimi kołami, że skrajnie prawy punkt zbioru $K \cap K'$ ma współrzędną x mniejszą niż $X.x$. To oczywiście oznacza, że wszystkie punkty zbioru $K \cap K'$ mają współrzędne x mniejsze od $X.x$, a tym samym nie mogą należeć do zbioru S. Dochodzimy do sprzeczności z założeniem, iż S ma część wspólną z K i K'. ■

W ten sposób sprawdzanie, czy zbiór S (dość niewygodny w obliczeniach) jest pusty, sprowadziliśmy do sprawdzenia, czy należy do niego jeden konkretny punkt — skrajnie lewy punkt zbioru P. Co ważne, zbiór P jest znacznie łatwiejszy do opisania i do wyliczenia niż zbiór S, a przy tym można go łatwo aktualizować po dodaniu nowego koła do zbioru $\mathcal{K}$.

Algorytm

Niech $K_1, K_2, \ldots, K_n$ będą kolejnymi kołami opisanymi w danych. Ponadto przez $\mathit{prawyPunkt}(K, K')$ oznaczmy funkcję zwracającą jako wartość skrajnie prawy punkt zbioru $K \cap K'$ — jeśli koła nie mają wspólnych punktów, to przyjmiemy, że funkcja zwraca wartość *NULL*.

```
1:  procedure AkcjaKomandosów
2:    begin
3:      X:=prawyPunkt(K1,K2);
4:      if X=NULL then return 2;
5:      for i:=3 to n do
6:        for j:=1 to i−1 do
7:          Y = prawyPunkt(Ki, Kj);
8:          if Y=NULL then return i;
9:          if Y.x⩽X.x then X:=Y;
10:       for j:=1 to i do
11:         if X ∉ Kj return i;
12:     return NIE;
13:   end
```

Pozostaje jeszcze wyjaśnić, w jaki sposób znaleźć skrajnie prawy punkt części wspólnej dwóch kół. Wystarczy zauważyć, że jest to albo skrajnie prawy punkt któregoś z kół, albo któryś z punktów przecięcia okręgów będących brzegami zadanych kół. Potrafimy więc go znaleźć w czasie stałym. Stąd cały algorytm działa w czasie $O(n^2)$ i wymaga pamięci $O(n)$.

Rozwiązania zawodników

Większość rozwiązań nadesłanych do oceny opierała się na niepoprawnych algorytmach. W wielu programach sprawdzano jedynie, czy w danym momencie każda para kół ma niepuste przecięcie. Warunek ten — jakkolwiek jest warunkiem koniecznym na istnienie niepustego przecięcia wszystkich kół — nie jest dostateczny. Można łatwo narysować już trzy koła, z których każde dwa przecinają się, lecz część wspólna wszystkich trzech jest pusta. Rozwiązania bazujące na tej błędnej idei były oceniane na zero punktów.

Testy

Rozwiązania sprawdzane były przy użyciu 22 wygenerowanych losowo zestawów danych testowych. Testy *akc5a.in* i *akc5b.in*, jak również *akc20a.in* i *akc20b.in* były zgrupowane. Testy można podzielić na kilka grup:

- *akc1.in* – *akc5.in* — testy poprawnościowe
- *akc6.in* – *akc9.in* — testy wydajnościowe pozwalające odrzucić programy działające w czasie $O(n^4)$ i dłuższym
- *akc10.in* – *akc20.in* — testy wydajnościowe pozwalające odrzucić programy działające w czasie $O(n^3)$

Nazwa	n
akc1.in	23
akc2.in	43
akc3.in	58
akc4.in	70
akc5a.in	70
akc5b.in	100
akc6.in	150
akc7.in	176
akc8.in	250
akc9.in	360
akc10.in	500

Nazwa	n
akc11.in	788
akc12.in	943
akc13.in	1 008
akc14.in	1 000
akc15.in	1 363
akc16.in	1 503
akc17.in	1 703
akc18.in	2 000
akc19.in	2 000
akc20a.in	2 000
akc20b.in	2 000