Zawody I stopnia

opracowania zadań

Grzegorz Jakacki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Gra w wielokąty

W grze w wielokąty uczestniczy dwóch graczy. Rekwizytem jest wielokąt wypukły o n wierzchołkach podzielony przez $n-3$ przekątne na $n-2$ trójkąty. Żadne dwie z tych przekątnych nie przecinają się poza wierzchołkami wielokąta. Jeden z trójkątów jest czarny, a pozostałe — białe. Gracze na przemian odcinają od wielokąta po jednym trójkącie, za każdym razem przecinając wielokąt wzdłuż jednej z danych przekątnych. Gracz, który odetnie czarny trójkąt, wygrywa.

PRZYPOMNIENIE: Wielokąt jest wypukły, *jeśli odcinek łączący dowolne dwa jego punkty jest całkowicie zawarty w wielokącie.*

Zadanie

Napisz program, który:

- *czyta z pliku tekstowego* `GRA.IN` *opis rekwizytu do gry,*
- *sprawdza, czy gracz rozpoczynający grę ma strategię wygrywającą,*
- *zapisuje wynik do pliku* `GRA.OUT`.

Wejście

Pierwszy wiersz pliku wejściowego `GRA.IN` *zawiera liczbę naturalną n, $4 \leqslant n \leqslant 50000$. Jest to liczba wierzchołków wielokąta. Wierzchołki wielokąta są ponumerowane kolejnymi liczbami od 0 do $n-1$, zgodnie z ruchem wskazówek zegara.*

Następnych $n-2$ wierszy zawiera opisy trójkątów w wielokącie. W wierszu o numerze $i+1$, $1 \leqslant i \leqslant n-2$, znajdują się trzy liczby naturalne a, b, c, oddzielone pojedynczymi odstępami. Są to numery wierzchołków i-tego trójkąta. Pierwszy trójkąt w ciągu jest czarny.

Wyjście

Plik wyjściowy `GRA.OUT` *powinien składać się z jednego wiersza zawierającego jedno słowo:*

- `TAK`, *jeśli gracz rozpoczynający grę ma strategię wygrywającą,*
- `NIE`, *jeśli nie ma.*

Przykład

Dla pliku wejściowego `GRA.IN`*:*

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

poprawną odpowiedzią jest plik tekstowy `GRA.OUT`*:*

```
TAK
```

ROZWIĄZANIE

Zadanie było jednym z najprostszych zadań prezentowanych kiedykolwiek na Olimpiadzie Informatycznej. Już pobieżna analiza przedstawionej gry pokazuje, że:

- jeżeli czarny trójkąt można odciąć w jednym ruchu, to wygrywa rozpoczynający grę,

- w przeciwnym przypadku istnienie strategii wygrywającej dla rozpoczynającego grę zależy wyłącznie od parzystości liczby wierzchołków rekwizytu. Strategia wygrywająca polega na unikaniu odcinania sąsiadów czarnego trójkąta.

Z uwagi na fakt, że wierzchołki wielokąta są ponumerowane kolejno, sprawdzenie, czy czarny trójkąt może zostać odcięty w pierwszym ruchu sprowadza się do ustalenia, czy jego wierzchołkami są trzy kolejne wierzchołki wielokąta. (UWAGA: należy pamiętać, że po wierzchołku nr $n-1$ następuje wierzchołek nr 0, zatem wierzchołki $n-1, 0, 1$ też uważamy za kolejne. Pamiętajmy też, że specyfikacja zadania nie określa porządku, w jakim podane są numery wierzchołków trójkąta w pliku wejściowym.)

Zauważmy, że znalezienie rozwiązania wymaga przeczytania co najwyżej dwóch pierwszych wierszy pliku wejściowego. Programy, które wczytywały cały plik wejściowy były skazane na niepowodzenie przy dużych testach z powodu przekroczenia limitu czasu.

TESTY

Do oceny rozwiązań zawodników użyto 29-ciu testów GRA0.IN–GRA28.IN. By zapobiec przyznawaniu punktów programom, które zawsze generują taki sam plik wyjściowy, testy GRA1.IN–GRA28.IN łączono w dwuelementowe grupy (test GRA$2i$.IN z testem GRA$(2i+1)$.IN). Punkty za każdy test w grupie były przyznawane jedynie wtedy, gdy oba testy zostały zaliczone.

Krzysztof Loryś
autor zadania, opracowanie

Radosław Szklarczyk
program wzorcowy

Monocyfrowe reprezentacje

*Niech K będzie cyfrą dziesiętną, różną od 0. K-*reprezentacją liczby całkowitej X *nazywamy wyrażenie arytmetyczne o wartości X, w którym wszystkie liczby (w postaci dziesiętnej) składają się wyłącznie z cyfry K. Dopuszczalnymi działaniami w takim wyrażeniu są: dodawanie, odejmowanie, mnożenie i dzielenie. Można też używać nawiasów okrągłych. Dzielenie może wystąpić tylko wtedy, gdy dzielnik dzieli dzielną bez reszty.*

Przykład

Każde z poniższych wyrażeń jest 5-reprezentacją liczby 12:

- *5 + 5 + (5 : 5) + (5 : 5)*
- *(5 + (5)) + 5 : 5 + 5 : 5*
- *55 : 5 + 5 : 5*
- *(55 + 5) : 5*

Długością *K-reprezentacji nazywamy liczbę wystąpień w niej cyfry K. W powyższym przykładzie dwie pierwsze reprezentacje mają długość 6, trzecia — długość 5, a czwarta — długość 4.*

Zadanie

Napisz program, który:

- *z pliku tekstowego* `MON.IN` *wczytuje cyfrę K oraz ciąg liczb,*
- *dla każdej liczby z tego ciągu sprawdza, czy istnieje jej K-reprezentacja nie dłuższa niż 8 i jeśli tak, to znajduje minimalną długość takiej reprezentacji,*
- *zapisuje wyniki do pliku tekstowego* `MON.OUT`.

Wejście

Pierwszy wiersz pliku wejściowego `MON.IN` *zawiera cyfrę K, $K \in \{1, \ldots, 9\}$. Drugi wiersz zawiera liczbę n, $1 \leqslant n \leqslant 10$. W następnych n wierszach znajduje się ciąg liczb naturalnych $a_1, \ldots, a_n$, $1 \leqslant a_i \leqslant 32000$ (dla $i = 1, \ldots, n$), po jednej liczbie w każdym wierszu.*

Wyjście

Plik `MON.OUT` *składa się z n wierszy. Wiersz nr i powinien zawierać:*

- *dokładnie jedną liczbę będącą minimalną długością K-reprezentacji a_i, o ile ta długość nie przekracza 8,*
- *jedno słowo* `NIE`*, jeżeli minimalna długość K-reprezentacji liczby a_i jest większa niż 8.*

Przykład

Dla pliku wejściowego `MON.IN`*:*

```
5
2
12
31168
```

poprawną odpowiedzią jest plik tekstowy `MON.OUT`*:*

```
4
NIE
```

ROZWIĄZANIE

W swej pierwotnej wersji zadanie różniło się od wersji ostatecznej dwoma istotnymi elementami:

- ograniczenie na długość K-reprezentacji było większe i wynosiło 10,
- wynikiem działania programu miało być wyrażenie o minimalnej długości, a nie tylko jego długość.

W niniejszym omówieniu odstąpimy nieco od tradycji prezentowania rozwiązań jedynie ostatecznych wersji zadań i, kierując się jego walorami dydaktycznymi, najpierw omówimy rozwiązanie pierwotnej wersji. Dopiero potem prześledzimy uproszczenia wynikające ze zmian wprowadzonych do treści zadania.

Zadanie w pierwotnej wersji

Zadanie nie jest skomplikowane algorytmicznie i główna trudność leży nie tyle w znalezieniu efektywnego algorytmu (ten daje narzucająca się metoda programowania dynamicznego), co w starannym doborze struktur danych.

Prosta strategia dynamiczna polega na generowaniu K-reprezentacji o coraz większych długościach. Niech $L_K(d)$ będzie zbiorem liczb naturalnych, których minimalna K-reprezentacja ma długość d. Oczywiście $L_K(1)$ zawiera tylko jeden element — liczbę K. Następnie, zakładając, że mamy obliczone zbiory $L_K(d')$, dla $1 \leqslant d' < d$, możemy obliczyć $L_K(d)$ korzystając z prostej własności podanej w poniższym fakcie.

Fakt 1: Liczba s należy do $L_K(d)$ wtedy i tylko wtedy, gdy

- $s = \underbrace{KK\ldots K}_{d}$ lub
- $\forall_{d'<d}\ s \notin L_K(d')$ oraz istnieją l i r takie, że
  - $s = l \odot r$, gdzie $\odot$ jest działaniem ze zbioru $\{+, -, *, /\}$ i
  - $l \in L_K(d_1)$ i $r \in L_K(d_2)$, dla pewnych d_1, d_2 takich, że $d_1 + d_2 = d$.

Struktura danych, w której będziemy pamiętać elementy zbiorów $L_K(d')$, dla $1 \leqslant d' \leqslant d$, powinna umożliwiać szybkie wykonywanie następujących operacji słownikowych:

- sprawdzenie, czy dana liczba s należy już do któregoś z wygenerowanych zbiorów,
- wstawienie s do zbioru;

a także:

- umożliwiać szybki dostęp do wszystkich elementów poszczególnych zbiorów,
- zawierać informacje pozwalające na szybkie odtworzenie minimalnej K-reprezentacji dla wybranej liczby.

Aby racjonalnie zaprojektować taką strukturę powinniśmy na wstępie oszacować liczności zbiorów $L_K(d)$. Niestety nie jest znany prosty wzór określający te wielkości. Jak zwykle w takich przypadkach, warto wyznaczyć je eksperymentalnie (to była jedna z przesłanek umieszczenia zadania w zestawie na pierwszy etap zawodów — w trakcie zawodów II i III stopnia nie ma czasu na przeprowadzanie eksperymentów). Okazuje się, zgodnie z intuicją, iż rozmiar $L_K(d)$ rośnie bardzo szybko (wykładniczo) wraz ze wzrostem

Rys. 1 Wzrost liczności zbiorów $L_K(d)$ dla $K = 8$. Łącznie do zbiorów $L_8(1), \ldots, L_8(9)$ należy 17857 liczb.

d	$\lvert L_8(d)\rvert$
1	1
2	4
3	13
4	39
5	118
6	355
7	1145
8	3762
9	12420

d (patrz rysunek 1). Dla nas użyteczne jest nie tyle wyznaczenie tempa wzrostu, co konkretnych wartości $|L_K(d)|$.

Wartość $K = 8$ w tabeli z rysunku 1 nie została wybrana przypadkowo. Okazuje się bowiem, że zachodzi następujący fakt:

Fakt 2: Dla $K \neq 8$, wartości $|L_K(d)|$ rosną wolniej niż dla $K = 8$.

Sprawdzenie tego faktu pozostawiamy czytelnikowi. Analizując tabelę możemy wyciągnąć kilka praktycznych wniosków.

Obserwacja 3: Lepiej nie pamiętać $L_K(10)$.

Na rysunku 1 nie podano mocy zbioru $L_8(10)$, ale ekstrapolując można przyjąć rozsądne założenie, że wynosi ona około 40000. Gdybyśmy chcieli zapamiętać wszystkie zbiory od $L_8(1)$ do $L_8(10)$, to struktura powinna być w stanie pomieścić około 60000 elementów, a więc na jeden element moglibyśmy przeznaczyć nie więcej niż 10-11 bajtów. Ponieważ w każdym elemencie struktury musimy przeznaczyć minimum 4 bajty na zapamiętanie liczby (w rzeczywistości liczby z $L_K(10)$ mogą przekraczać wartość pascalowego *MaxLongInt*), pozostanie nam nie więcej niż 7 bajtów — za mało nawet na dwa wskaźniki. W tej sytuacji trudno myśleć o utworzeniu sensownej struktury.

Ponieważ zbiór $L_K(10)$ nie jest nam potrzebny do wyznaczania dalszych zbiorów, możemy zrezygnować z zapamiętywania go. W zamian będziemy musieli jedynie dokonać drobnych modyfikacji programu tak, by przypadek $d = 10$ był odrębnie rozpatrywany.

Obserwacja 4: Czas nie powinien być zasobem krytycznym.

Prosty sposób wyznaczania zbioru $L_K(d)$ polega na rozpatrywaniu wszystkich par liczb, z których jedna należy do $L_K(d_1)$, a druga do $L_K(d_2)$, dla wszystkich d_1 i d_2 sumujących się do d. Takich par jest $\sum_{d_1=1}^{\lfloor d/2 \rfloor} |L_K(d_1)| \cdot |L_K(d-d_1)|$. W najgorszym przypadku ($K = 8$) musimy więc rozważyć 95203 pary, z czego tylko 25081 to pary służące do wyznaczenia $L_8(d)$ (dla $d \leqslant 9$). Wielkości te nie są szokujące, musimy jednak pamiętać, że:

- dla każdej pary będziemy obliczać sumę, różnicę, iloczyn i iloraz,
- dla par rozpatrywanych podczas obliczania $L_K(d)$, dla $d \leqslant 9$, co najmniej trzy z tych wyników (z pominięciem ilorazu w przypadkach, gdy nie jest on liczbą całkowitą) będziemy odszukiwać w strukturze.

Wydaje się, że naszą strukturę danych możemy zbudować w oparciu o drzewa binarnych przeszukiwań, np. mógłby to być las drzew: po jednym drzewie dla każdego zbioru $L_K(d)$. Jeśli w każdym węźle zapamiętamy liczbę typu *longint* oraz dwa wskaźniki na potomków, to pozostanie jeszcze sporo miejsca na informacje, które pozwolą na szybkie odtworzenie K-reprezentacji. Nieco gorzej może być z czasem wyszukiwania. Musimy liczyć się z koniecznością wykonania nawet 100000 wyszukiwań, z czego większość nastąpi w momencie obliczania $L_K(9)$. Wówczas wiele z nich będziemy musieli przeprowadzać we wszystkich dziewięciu drzewach. Co prawda nie są one zbyt duże (patrz tabela na rysunku 1), jednak łączny czas wyszukiwań może okazać się znaczny. Przykładowo, dla $K = 8$ można oczekiwać, że w średnim przypadku jedno wyszukiwanie będzie wymagać wykonania ponad 70 porównań.

Lepszym rozwiązaniem jest zapamiętanie wszystkich zbiorów $L_K(d)$ w jednym drzewie. Aby umożliwić szybki dostęp do elementów poszczególnych zbiorów możemy przeszyć drzewo i utworzyć z jego węzłów 9 list: po jednej na każdy zbiór. To będzie wymagać tylko jednego dodatkowego pola na wskaźnik w każdym węźle. Całe drzewo będzie teraz składać się z mniej niż 18000 węzłów i liczba porównań podczas wyszukiwania znacznie się zmniejszy. Niestety nadal może być zbyt duża. Nawet gdyby drzewo było doskonale zrównoważone, to średnia długość ścieżki wynosiłaby około 14, co przy liczbie wyszukiwań rzędu 100000 nakazuje ostrożność. W rzeczywistości może okazać się, że jest znacznie gorzej. Ponieważ do drzewa dołączamy elementy o coraz dłuższych K-reprezentacjach, należy oczekiwać, że statystycznie będą to coraz większe liczby i wystąpi tendencja do nadmiernego wzrostu prawych poddrzew.

Sytuację można poprawić stosując jeden z rodzajów drzew zrównoważonych (np. drzewa AVL). Proponujemy jednak inne, znacznie szybsze rozwiązanie, oparte na zastosowaniu funkcji haszujących. Główną składową struktury są listy, których początki przechowywane są w tablicy, powiedzmy o nazwie tab i rozmiarze m. Wartością $tab[i]$ jest wskaźnik na początek listy, w której pamiętane będą te liczby ze zbiorów $L_K(d)$, dla których funkcja haszująca h przyjmuje wartość i. Załóżmy, że chcemy sprawdzić, czy w strukturze znajduje się liczba s. Najpierw musimy obliczyć wartość $h(s)$, a następnie przeszukać listę, której początek wskazywany jest przez $tab[h(s)]$. Koszt takich operacji jest więc zależny od szybkości obliczania wartości funkcji haszującej oraz od długości powstałych list. Ponieważ oczekiwana długość list jest równa stosunkowi liczby elementów przechowywanych w strukturze do rozmiaru tablicy tab, więc w naszym interesie leży, by rozmiar m tablicy był możliwie duży. Jako funkcji haszującej możemy użyć stosunkowo szybko obliczalnej funkcji $h(x) = x \textbf{ mod } m$. Efektywność takiego rozwiązania najlepiej zilustrować na przykładzie (zob. rysunek 2).

Rys. 2 Rozkład długości list powstałych po umieszczeniu w strukturze wszystkich liczb z $L_8(d)$ (dla $d = 1, \ldots, 9$). Jako m przyjęto 15013.

Długość listy (k)	**Liczba list o długości k**
0	4005
1	5986
2	3520
3	1208
4	263
5	31

Zauważmy, że nie powstała ani jedna lista zawierająca 6 lub więcej elementów. Tak więc każda operacja wyszukiwania elementu w strukturze będzie wymagać nie więcej niż 5 porównań. Jeszcze bardziej zaskakująca dla czytelnika może być średnia liczba porównań przypadająca na jedno wyszukiwanie. Wynosi ona ok. 1,37, wobec 14 w drzewie doskonale zrównoważonym!

Podobnie jak w przypadku drzew, listy można przeszyć, by możliwy był sekwencyjny dostęp do elementów poszczególnych zbiorów $L_K(d)$.

Pozostaje jeszcze do omówienia sposób odtwarzania K-reprezentacji. Wstawiając liczbę do struktury zapamiętujemy informację o tym, na jakiej podstawie dokonujemy wstawienia: czy na podstawie pierwszego punktu faktu 1, czy też na podstawie drugiego. W tym drugim przypadku zapamiętujemy informację o działaniu i jego argumentach (w rzeczywistości wystarcza informacja o działaniu i większym z argumentów). Te informacje pozwalają na odtworzenie K-reprezentacji przy pomocy prostej procedury rekurencyjnej.

Oczywiście można poczynić jeszcze sporo usprawnień przyspieszających działanie algorytmu, jak na przykład przerywanie generowania zbiorów $L_K(d)$ w momencie umieszczenia w strukturze wszystkich liczb a_i (patrz treść zadania). Nie omawiamy ich, pozostawiając to inwencji czytelnika.

Zadanie w ostatecznej wersji

Ograniczenie długości K-reprezentacji do 8 radykalnie upraszcza zadanie. Można sprawdzić, że dla każdego K, liczb mających K-reprezentację o długości nie większej niż 8 jest mniej niż 5500. Dzięki temu dowolna ze struktur rozpatrywanych wyżej pozwala na zadowalająco efektywną implementację algorytmu. Z kolei zrezygnowanie z konieczności wypisywania K-reprezentacji na rzecz wypisywania tylko ich długości pozwala na zastosowanie tricku polegającego na tzw. *preprocessingu*, który jest często używany podczas zawodów programistycznych. Polega on na uprzednim utworzeniu, często kosztem żmudnych obliczeń, struktur danych i umieszczeniu ich jako stałych w programie. Dzięki temu można znacznie przyspieszyć działanie programu, a często nawet sprowadzić je do odczytania odpowiednich informacji ze struktury.

W przypadku naszego zadania bazujemy na fakcie, iż w przedziale [1..32000] nie ma zbyt wielu liczb posiadających K-reprezentacje o długości nie większej od 8. Ilustruje to poniższa tabela.

K	$\sum_{d=1}^{8} \lvert L_K(d) \cap [1..32000] \rvert$
1	418
2	1382
3	2066
4	2763
5	2623
6	2950
7	3210
8	2937
9	2730

Możemy utworzyć 9 tablic, po jednej dla każdego K, których elementami będą rekordy typu:

```
record
    liczba : integer;
    długość : byte
end
```

W rekordach K-tej tablicy, w polu *liczba*, pamiętana jest liczba posiadająca K-reprezentację o długości nie większej od 8, a w polu *długość* minimalna długość takiej reprezentacji. Tablice te możemy umieścić w sekcji stałych programu. Wówczas jego działanie sprowadzi się do odszukania w odpowiedniej tablicy liczb a_i. Jeśli zadbamy, by rekordy w tablicy były uporządkowane rosnąco względem pola *liczba*, to naturalnym będzie zastosowanie wyszukiwania binarnego.

Zauważmy jeszcze na koniec, że na wszystkie tablice potrzebujemy w sumie zaledwie 63237 bajtów (21079 trzybajtowych rekordów).

TESTY

Do oceny rozwiązań użyto 13-tu testów MON0.IN-MON12.IN. Testy MON0.IN i MON1.IN są testami poprawnościowymi. Testy MON2.IN-MON10.IN zawierają pytania o K-reprezentacje dla kolejnych wartości K. I tak MON2.IN zawiera pytanie o 1-reprezentację, MON3.IN — pytania o 2-reprezentacje, …, MON10.IN — pytania o 9-reprezentacje. Celem tych testów była, poza sprawdzeniem poprawności, ocena efektywności programów. Dlatego liczby, o które pytamy w kolejnych testach z tej grupy, są coraz większe i mają coraz dłuższe minimalne K-reprezentacje. W testach MON11.IN i MON12.IN pytamy o 9-reprezentacje liczb jeszcze większych. Dla niektórych z tych liczb odpowiedź jest negatywna.

Wojciech Guzicki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Muszkieterowie

W czasach Ludwika XIII i jego potężnego ministra, kardynała Richelieu, w gospodzie pod Pełnym Antałkiem raczyło się jadłem i winem n muszkieterów. Wina było pod dostatkiem, a że muszkieterowie byli skorzy do zwady, doszło do wielkiej awantury, w której każdy muszkieter obraził wszystkich pozostałych.

Musiało dojść do pojedynków, ale kto miał się z kim pojedynkować i w jakiej kolejności? Uzgodnili (po raz pierwszy od awantury zrobili coś zgodnie), że staną na okręgu w ustalonej kolejności i będą ciągnąć losy. Wylosowany muszkieter toczył pojedynek ze swoim sąsiadem z prawej. Przegrany „odpadał z gry", a dokładniej — służący wynosili jego zwłoki. Sąsiadem wygrywającego stawał się następny muszkieter, stojący za przegranym.

Po latach, gdy historycy czytali wspomnienia zwycięzcy, spostrzegli, że ostateczny wynik zależał w istotny sposób od kolejności wylosowanych pojedynków. Zauważyli bowiem, że dotychczasowe treningi fechtunku pokazywały, kto z kim mógł wygrać pojedynek. Okazało się, że — mówiąc językiem matematycznym — relacja „A wygrywa z B" nie była przechodnia! Mogło się zdarzyć, że muszkieter A walczył lepiej od muszkietera B, B lepiej od C, a C lepiej od A. Oczywiście wśród takiej trójki wybór pierwszego pojedynku decydował o ostatecznym wyniku! Jeśli pierwsi będą walczyć A i B, to ostatecznie wygra C, ale jeśli pierwszymi będą B i C, to ostatecznie wygra A. Historycy, zafascynowani tym odkryciem, postanowili sprawdzić, którzy muszkieterowie mogli przeżyć — od tego zależał przecież los Francji, a wraz z jej losem, los całej cywilizowanej Europy!

Zadanie

Na okręgu stoi n osób, ponumerowanych kolejno liczbami od 1 do n. Osoby te toczą $n-1$ pojedynków. W pierwszej rundzie jedna z tych osób, powiedzmy o numerze i — toczy pojedynek z sąsiadem z prawej, tzn. z osobą o numerze $i+1$ (lub, jeśli $i=n$, to z osobą o numerze 1). Przegrany odpada z gry, sąsiadem zwycięzcy staje się następna w kolejności osoba. Dana jest też tabela możliwych wyników pojedynków w postaci macierzy A. Jeśli $A_{i,j}=1$, to osoba o numerze i zawsze wygrywa z osobą o numerze j. Jeżeli $A_{i,j}=0$, to osoba o numerze i przegrywa z osobą o numerze j. Mówimy, że osoba k może wygrać *zawody, jeżeli istnieje taki ciąg $n-1$ losowań, w wyniku którego będzie ona zwycięzcą ostatniego pojedynku.*

Napisz program, który:

- *wczyta z pliku tekstowego* `MUS.IN` *macierz A,*
- *wyznaczy numery osób, które mogą wygrać zawody,*
- *zapisze je do pliku tekstowego* `MUS.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `MUS.IN` *dana jest liczba całkowita n spełniająca nierówność $3 \leqslant n \leqslant 100$. W każdym z następnych n wierszy znajduje się jedno słowo złożone z n znaków* `0` *lub* `1`*. Znak stojący na j-tym miejscu w i-tym z tych wierszy oznacza wartość $A_{i,j}$ (tzn. jeśli tym znakiem jest jedynka, to $A_{i,j}=1$, a jeśli zero, to $A_{i,j}=0$). Oczywiście $A_{i,j}=1-A_{j,i}$ dla $i \neq j$. Przyjmujemy, że $A_{i,i}=1$, dla każdego i.*

Wyjście

W pierwszym wierszu pliku wyjściowego `MUS.OUT` *należy zapisać liczbę m tych osób, które mogą wygrać zawody. W następnych m wierszach należy zapisać numery tych osób w kolejności rosnącej, po jednym numerze w wierszu.*

Przykład

Dla pliku wejściowego `MUS.IN`*:*

```
7
1111101
0101100
0111111
```

```
0001101
0000101
1101111
0100001
```

kolejność pojedynków: 1–2, 1–3, 5–6, 7–1, 4–6, 6–1 daje ostateczną wygraną osobie o numerze 6. Można sprawdzić, że jeszcze tylko osoby o numerach 1 i 3 mogą wygrać zawody. Zatem plik wyjściowy `MUS.OUT` *powienien wyglądać następująco:*

```
3
1
3
6
```

ROZWIĄZANIE

Wprowadźmy oznaczenia (i oraz j są tu liczbami naturalnymi z przedziału od 1 do n):

$$i \prec j \Leftrightarrow \text{ osoba } j \text{ wygrywa z osobą } i,$$

oraz

$$i \oplus j = \begin{cases} i+j & \text{gdy } i+j \leqslant n \\ i+j-n & \text{gdy } i+j > n. \end{cases}$$

Zauważmy najpierw, że algorytm polegający na sprawdzeniu wszystkich kolejności losowań będzie działał w czasie rzędu $n!$. Dla większych n taki program nie daje szans na uzyskanie wyniku — czas jego działania będzie nieprawdopodobnie długi. Jednak dla n mniejszych od 10 taki program może dość szybko dać odpowiedź i, gdy brakuje nam lepszego pomysłu, powinniśmy oczywiście taki program napisać.

Rozwiązanie „autorskie" polega na zauważeniu najpierw, że wystarczy rozważyć wszystkie pojedynki rozgrywane na odcinkach, a nie na okręgu, tzn. wśród osób o numerach

$$i, i \oplus 1, i \oplus 2, \ldots, i \oplus (n-1)$$

dla $i = 1, 2, \ldots, n$. W takim przypadku nie uznajemy osób i oraz $i \oplus (n-1)$ za sąsiadów. Teraz dla każdych i, j znajdujemy zbiór $Z_{i,j}$ osób, które mogą wygrać zawody na odcinku

$$i, i \oplus 1, \ldots, j.$$

Na przykład $Z_{2,6}$ to zbiór osób, które mogą wygrać zawody na odcinku $2, 3, 4, 5, 6$, a zbiór $Z_{n-3,2}$ to zbiór osób, które mogą wygrać pojedynek na odcinku $n-3, n-2, n-1, n, 1, 2$. Zauważmy, że szukane rozwiązanie jest sumą zbiorów $Z_{i,i\oplus(n-1)}$, dla $1 = 1, 2, \ldots, n$. Zbiory $Z_{i,j}$ znajdujemy „przekątnymi". Najpierw kładziemy $Z_{i,i} = \{i\}$. Następnie dla kolejnych s znajdujemy zbiory $Z_{i,i\oplus s}$. I tak dla $s = 1$ mamy

$$Z_{i,i\oplus 1} = \begin{cases} \{i\} & \text{gdy } A_{i,i\oplus 1} = 1 \\ \{i \oplus 1\} & \text{gdy } A_{i,i\oplus 1} = 0. \end{cases}$$

Dla większych s korzystamy z wyznaczonych dotychczas zbiorów Z. Można to zrobić dwiema metodami:

(1) Chcemy wyznaczyć $Z_{i,j}$. Wtedy dla dowolnego k znajdującego się na odcinku $i, \ldots, j$ rozpatrujemy zbiory $X = Z_{i,k}$ oraz $Y = Z_{k\oplus 1,j}$. Następnie dla każdego $x \in X$ i $y \in Y$ rozważamy wynik pojedynku x z y i zwycięzcę dołączamy do zbioru $Z_{i,j}$. Ten sposób obliczania zbioru $Z_{i,j}$ wymaga czasu $O(l_{i,j}^3)$, gdzie $l_{i,j}$ jest długością odcinka $i, i \oplus 1, \ldots, j$. Ponieważ musimy wyznaczyć n^2 takich zbiorów, więc całkowita złożoność tego algorytmu wyniesie $O(n^5)$.

(2) Dla każdego a z odcinka $i, \ldots, j$ sprawdzamy, czy a należy do $Z_{i,j}$. Dla $a = i$ korzystamy z następującego prostego faktu:

- $a \in Z_{i,j}$ wtedy i tylko wtedy, gdy istnieje b na odcinku $i \oplus 1, \ldots, j$ takie, że $b \prec a$ oraz $b \in Z_{i\oplus 1,j}$.

Dla $a = j$ korzystamy z podobnego faktu. Wreszcie dla pozostałych a korzystamy z kolejnego faktu:

- $a \in Z_{i,j}$ wtedy i tylko wtedy, gdy $a \in Z_{i,a}$ oraz $a \in Z_{a,j}$.

Ten sposób wyznaczania $Z_{i,j}$ wymaga czasu $O(l_{i,j})$ i cały algorytm będzie miał złożoność $O(n^3)$.
Opisany algorytm działa w czasie $O(n^3)$ i wymaga dość dużej pamięci: musimy zapamiętać n^2 zbiorów, z których każdy ma co najwyżej 100 elementów. Istnieje inny algorytm, działający również w czasie $O(n^3)$, ale wymagający znacznie mniej pamięci. Jest on praktycznie dużo szybszy od algorytmu opisanego wyżej.

W tym algorytmie również rozpatrujemy pojedynki odbywające się na odcinkach od i do j. W tablicy kwadratowej wymiaru $n \times n$ zapisujemy, czy możliwy jest taki przebieg pojedynków, by muszkieterowie o numerach i oraz j stanęli obok siebie (tzn. taki przebieg, że wszyscy muszkieterowie rozdzielający muszkieterów i oraz j giną). Jeśli taki przebieg pojedynków jest możliwy, to kładziemy $T[i,j] = \textbf{true}$. Oczywiście $T[i,j]$ i $T[j,i]$ oznaczają co innego: $T[i,j] = \textbf{true}$ oznacza, że giną wszyscy muszkieterowie z odcinka od i do j; $T[j,i]$ oznacza, że giną wszyscy muszkieterowie z odcinka od j do i (w obu przypadkach bez muszkieterów i oraz j). Tablicę T wypełniamy również „przekątnymi”. Najpierw kładziemy $T[i, i \oplus 1] = \textbf{true}$. Następnie, jeśli j nie jest postaci $i \oplus 1$, to kładziemy $T[i,j] = \textbf{true}$ wtedy i tylko wtedy, gdy istnieje numer k w odcinku od i do j taki, że $T[i,k] = T[k,j] = \textbf{true}$ oraz któryś z muszkieterów i lub j potrafi wygrać z muszkieterem k. Po zapełnieniu całej tablicy szukamy zbioru możliwych zwycięzców. Jeśli może zostać tylko dwóch muszkieterów i oraz j (czyli mamy jednocześnie $T[i,j] = T[j,i] = \textbf{true}$), to zwycięzca pojedynku i z j jest dołączany do zbioru zwycięzców.

TESTY

Do oceny rozwiązań zawodników użyto 20-tu testów MUS0.IN–MUS19.IN. Test MUS0.IN jest testem z treści zadania. Testy można podzielić na sześć typów:

(1) test klasyczny: każdy muszkieter może wygrać, muszkieter o większym numerze wygrywa, gdy różnica numerów jest parzysta (zakładamy tu, że liczba n jest nieparzysta);

(2) test rosnący: muszkieter o większym numerze zawsze wygrywa z muszkieterem o mniejszym numerze;

(3) test losowy: tablica wyników pojedynków wygenerowana losowo, każdy może wygrać;

(4) test prawie losowy: tablica wyników wygenerowana losowo, nie każdy może wygrać;

(5) splot dwóch testów: z dwóch testów (jeden dla k muszkieterów, drugi dla l muszkieterów) robimy trzeci (dla $k \cdot l$ muszkieterów) dzieląc muszkieterów na k grup po l muszkieterów. Każdy muszkieter dostaje dwa numery: numer grupy i numer w grupie. Jeśli muszkieterowie i oraz j mają ten sam numer grupy, to wynik jest taki sam, jak wynik walki w teście drugim muszkieterów o takich samych numerach, co numery i i j w tej grupie. Jeśli mają inne numery grup, to wynik jest taki, jak wynik walki muszkieterów w teście pierwszym o takich samych numerach, co numery grup muszkieterów i i j.

(6) testy, w których wszyscy muszkieterowie mogą być zwycięzcami, ale jeden wybrany muszkieter może wygrać w jeden tylko sposób (jest tylko jeden ciąg pojedynków, którego zwycięzcą jest ten wybrany muszkieter).

Testy od MUS1.IN do MUS19.IN były wybrane w następujący sposób:

- MUS1.IN — klasyczny dla 3 muszkieterów,
- MUS2.IN — rosnący dla 3 muszkieterów,
- MUS3.IN — klasyczny dla 39 muszkieterów,
- MUS4.IN — splot: rosnący dla 2 muszkieterów z klasycznym dla 3 muszkieterów (3 zwycięzców),
- MUS5.IN — klasyczny dla 75 muszkieterów,
- MUS6.IN — splot: prawie losowy dla 3 muszkieterów z klasycznym dla 3 muszkieterów (3 zwycięzców),

- MUS7.IN — klasyczny dla 99 muszkieterów,
- MUS8.IN — prawie losowy dla 10 muszkieterów (9 zwycięzców),
- MUS9.IN — splot: klasyczny dla 5 muszkieterów z prawie losowym dla 10 muszkieterów (45 zwycięzców),
- MUS10.IN — splot: losowy dla 15 muszkieterów z prawie losowym dla 5 muszkieterów (71 zwycięzców),
- MUS11.IN — splot: klasyczny dla 5 muszkieterów z prawie losowym dla 20 muszkieterów (95 zwycięzców),
- MUS12.IN — splot: prawie losowy dla 9 muszkieterów z losowym dla 10 muszkieterów (80 zwycięzców),
- MUS13.IN — splot: klasyczny dla 10 muszkieterów z rosnącym dla 10 muszkieterów (9 zwycięzców),
- MUS14.IN — jedna permutacja dla 10 muszkieterów,
- MUS15.IN — prawie losowy dla 8 muszkieterów (6 zwycięzców),
- MUS16.IN — jedna permutacja dla 15 muszkieterów,
- MUS17.IN — prawie losowy dla 8 muszkieterów (6 zwycięzców),
- MUS18.IN — jedna permutacja dla 20 muszkieterów,
- MUS19.IN — prawie losowy dla 8 muszkieterów (5 zwycięzców).

SYMULACJE LOSOWE

Warto jeszcze zwrócić uwagę na to, że można próbować wyznaczyć zbiór możliwych zwycięzców za pomocą symulacji losowych. Losujemy wiele możliwych przebiegów gry i za każdym razem znajdujemy zwycięzcę. Symulacja jednej gry wymaga n losowań. Trzeba tylko opracować dobry sposób usuwania pokonanych muszkieterów. W programie MUS-LOS.PAS został zastosowany najprostszy sposób: usuwanie elementu z tablicy. W tym celu wykorzystano typ *string*, w którym procedura *Delete* usuwa dany wyraz bardzo szybko. Można również utworzyć drzewo binarne złożone ze wszystkich muszkieterów i w czasie logarytmicznym znajdować i usuwać pokonanego muszkietera. Czas symulacji przebiegu jednych zawodów jest zatem rzędu $O(n^2)$ lub $O(n \log n)$, można więc dość szybko przeprowadzić nawet kilka tysięcy takich symulacji. Program symulacyjny nie daje zawsze odpowiedzi poprawnej. Jego istotną zaletą jest szybkość oraz to, że uzyskany zbiór zwycięzców jest podzbiorem prawdziwego zbioru zwycięzców. Nie jest możliwe błędne zakwalifikowanie kogoś jako zwycięzcy, co mogłoby się zdarzyć przy innych błędnych algorytmach.

Taki program symulacyjny może dać prawidłową odpowiedź tylko wtedy, gdy każdy muszkieter, który może być zwycięzcą, wygrywa w dostatecznie wielu rozgrywkach (tzn. istnieje dostatecznie wiele ciągów kolejnych losowań, w wyniku których zostaje on zwycięzcą). Chodzi o to, by prawdopodobieństwo wygrania było dość duże. Popatrzmy na kilka przykładów.

W teście MUS8.IN na łączną liczbę 1814400 ciągów losowań (zauważmy, że liczba takich ciągów jest równa $(n!/2)$: losujemy najpierw jednego muszkietera z n, potem jednego z $n-1$ itd.; nie musimy losować, gdy zostanie dwóch muszkieterów) poszczególni muszkieterowie wygrywają następującą liczbę razy:

Nr	**Liczba wygranych**	**Częstość**
1	96178	5,30%
2	95334	5,25%
3	572743	31,57%
4	0	0,00%
5	525663	28,97%
6	74312	4,10%
7	54026	2,98%
8	125756	6,93%
9	194912	10,74%
10	75476	4,16%

Test MUS8.IN wymagał tylko 44 symulacji. Niektóre testy wymagały jednak znacznie większej liczby symulacji. Na przykład test MUS15.IN, w którym muszkieter o numerze 1 wygrywał tylko w 36 przypadkach na łączną liczbę 20160 (tj. tylko w 0,18% przypadków), wymagał już 700 symulacji. Te symulacje zostały wykonane bardzo szybko: 0,11 sekundy na procesorze 386 (33MHz). Podobnie testy MUS17.IN i MUS19.IN wymagały tylko, odpowiednio, 305 i 491 symulacji. Test MUS3.IN wymagał tylko 191 symulacji (0,33 sek.), test MUS5.IN tylko 273 symulacji (1,10 sek.), test MUS7.IN tylko 711 symulacji (3,18 sek.), natomiast test MUS9.IN aż 9124 symulacji (14,01 sek.). Testy MUS14.IN, MUS16.IN i MUS18.IN zostały zaprojektowane specjalnie w ten sposób, by symulacja losowa nie mogła dać poprawnego wyniku. W tych testach jeden z muszkieterów może wygrać tylko w jeden sposób (zachęcam Czytelnika do znalezienia macierzy wyników pojedynków, która by to gwarantowała!). Oczywiście prawdopodobieństwo trafienia akurat na ten jeden sposób jest bardzo małe. Na przykład w teście MUS14.IN dziewięciu zwycięzców zostało znalezionych dopiero po 234119 symulacjach (ok. 55 sek.). W tym teście muszkieter o numerze 1 wygrywa tylko w jednym przypadku (na 1814400 wszystkich przypadków), a muszkieter o numerze 2 w 8 przypadkach. W teście MUS16.IN dziewięciu zwycięzców (na piętnastu) znaleziono dopiero po 514870 symulacjach (ponad 3 minuty). Podobnie zachowywały się niektóre duże testy: MUS10.IN wymagał 136355 symulacji (prawie 6 minut); w testach MUS11.IN i MUS12.IN kilkaset tysięcy symulacji (kilkanaście minut) nie wystarczyło do znalezienia wszystkich zwycięzców.

Taki sposób rozwiązania zadania, mimo iż nie jest w pełni poprawny, powinien być brany pod uwagę. Istnieje wiele problemów, których nie umiemy rozwiązać w inny sposób niż przez symulacje losowe. Istnieją też problemy, w których rozwiązanie znajdujemy w sposób losowy, a o poprawności znalezionego losowania możemy się przekonać w sposób deterministyczny. Przykładem takiego zadania, mającego duże znaczenie praktyczne, jest poszukiwanie tzw. niereszt kwadratowych (zachęcam Czytelnika do dowiedzenia się z podręcznika teorii liczb, co to są niereszty kwadratowe!). Przykładem bliższym nam może być zadanie „Jedynki i zera” z II Olimpiady Informatycznej. Chodziło w nim o to, by dla danej liczby n znaleźć jej wielokrotność mającą w zapisie dziesiętnym co najwyżej 100 cyfr: zer i jedynek. Jednym ze sposobów rozwiązania jest generowanie losowe liczb 100-cyfrowych składających się tylko z zer i jedynek i dzielenie ich przez n. Z chwilą, gdy uzyskamy resztę 0, kończymy dzielenie, jeśli nie uzyskamy zera, to losujemy następną liczbę (warto rozważyć oddzielnie przypadek liczb składających się z samych jedynek). Okazuje się, że na ogół wystarczyło wygenerować stosunkowo niewiele takich liczb (najwyżej ok. 3000). Program tak napisany działał szybko i dawał poprawną odpowiedź, gdy odpowiednia wielokrotność została znaleziona. Trudnym problemem jest jednak ustalenie maksymalnej liczby losowań.

Powróćmy jeszcze na chwilę do zadania o muszkieterach. Kusząca jest hipoteza:

$$\text{jeśli } a \text{ może wygrać zawody oraz } a \prec b, \text{ to } b \text{ też może wygrać zawody,}$$

która mogłaby znacznie przyspieszyć wyznaczenie szukanego zbioru za pomocą symulacji. Istnieje jednak kontrprzykład na tę hipotezę i zachęcam Czytelnika do znalezienia go.

Bogdan S. Chlebus
autor zadania, opracowanie

Marek Pawlicki
program wzorcowy

Puste prostopadłościany

Prostopadłościan nazwiemy regularnym*, gdy:*

- *jednym z jego wierzchołków jest punkt o współrzędnych $(0,0,0)$,*
- *krawędzie wychodzące z tego wierzchołka leżą na dodatnich półosiach układu współrzędnych,*
- *krawędzie te mają długości nie większe niż 10^6.*

Dany jest zbiór A punktów przestrzeni, których wszystkie współrzędne są całkowite i należą do przedziału $[1..10^6]$. Szukamy prostopadłościanu regularnego o maksymalnej objętości, który w swoim wnętrzu nie zawiera żadnego punktu ze zbioru A. Punkt należy do wnętrza prostopadłościanu *jeżeli jest punktem prostopadłościanu, ale nie jego ściany.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `PUS.IN` *współrzędne punktów ze zbioru A,*
- *wyznaczy jeden z prostopadłościanów regularnych o maksymalnej objętości, nie zawierający w swoim wnętrzu punktów ze zbioru A,*
- *zapisze wynik do pliku tekstowego* `PUS.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `PUS.IN` *znajduje się jedna całkowita, nieujemna liczba n, $n \leqslant 5000$, będąca liczbą elementów zbioru A.*

W kolejnych n wierszach pliku `PUS.IN` *znajdują się trójki liczb całkowitych z przedziału $[1..10^6]$ będące współrzędnymi (odpowiednio x, y i z) punktów ze zbioru A. Liczby w wierszu pooddzielane są pojedynczymi odstępami.*

Wyjście

W jedynym wierszu pliku wyjściowego `PUS.OUT` *powinny znaleźć się trzy liczby całkowite oddzielone pojedynczymi odstępami, będące współrzędnymi (odpowiednio x, y i z) tego wierzchołka znalezionego prostopadłościanu regularnego, który ma wszystkie współrzędne dodatnie.*

Przykład

Dla pliku wejściowego `PUS.IN`*:*

```
4
3 3 300000
2 200000 5
90000 3 2000
2 2 1000
```

poprawną odpowiedzią jest plik tekstowy `PUS.OUT`*:*

```
1000000 200000 1000
```

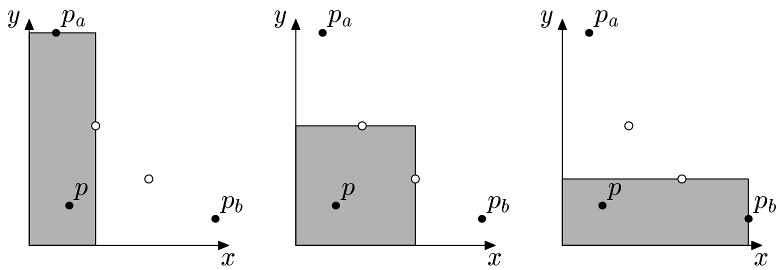
ROZWIĄZANIE

Regularny prostopadłościan, którego wnętrze nie zawiera punktów ze zbioru A, nazywamy *pustym*. Pusty prostopadłościan jest *maksymalny*, gdy nie jest właściwie zawarty w żadnym innym regularnym pustym prostopadłościanie. W trakcie obliczeń, na zmiennej *maxobj* pamiętamy największą objętość maksymalnego pustego prostopadłościanu spośród rozpatrywanych dotychczas, a jego wierzchołek o wszystkich współrzędnych różnych od 0 przechowujemy na zmiennej *pkt*. Uaktualnianie tych zmiennych polega na porównaniu wartości *maxobj* z objętością rozpatrywanego właśnie prostopadłościanu, i ewentualnym przypisaniu im nowych wartości, gdy badany prostopadłościan ma większą objętość. W trakcie wykonywania algorytmu będziemy odwoływali się do pewnego zbioru punktów $B \subseteq A$, który nazywamy *brzegiem*. Punkty należące do brzegu pamiętamy w porządku rosnącym względem współrzędnej x i jednocześnie w porządku malejącym względem współrzędnej y. Niech $p_1, \ldots, p_n$ będą punktami zbioru A uporządkowanymi niemalejąco względem współrzędnej z. Dla punktu p_i, niech p_i^x, p_i^y oraz p_i^z będą jego współrzędnymi.

Punkty A przeglądamy kolejno, zaczynając od p_1. Najpierw ustalamy brzeg na $B = \{p_1\}$. Maksymalny pusty prostopadłościan o podstawie $[0..10^6] \times [0..10^6]$ ma wysokość p_1^z.* Zapamiętujemy objętość tego prostopadłościanu na zmiennej *maxobj*, a jego wierzchołek o wszystkich współrzędnych różnych od 0 na zmiennej *pkt*. Wyższe puste prostopadłościany muszą mieć albo współrzędne x wnętrza nie większe od p_1^x albo współrzędne y wnętrza nie większe od p_1^y. Rozważmy punkt p_2. Jeżeli $p_2^x > p_1^x$ oraz $p_2^y > p_1^y$, to punkt p_2 nie może leżeć na ścianie żadnego pustego prostopadłościanu o większej objętości — zatem nie ma wpływu na rozwiązanie. Jeżeli $p_2^x < p_1^x$ oraz $p_2^y < p_1^y$ to punkt p_2 zastępuje p_1 w tym sensie, że każdy prostopadłościan o wysokości większej niż p_2^z musi mieć albo współrzędne x wnętrza nie większe od p_2^x, albo współrzędne y wnętrza nie większe od p_2^y. Kładziemy zatem $B = \{p_2\}$. Jednocześnie obliczamy objętość dwóch maksymalnych pustych prostopadłościanów, które mają wysokość p_2^z i dla których p_1 leży na jednej ze ścian. Jeżeli objętość któregoś z nich jest większa niż wartość *maxobj*, to przypisujemy nowe wartości *maxobj* i *pkt*. Jeżeli $p_2^x \leqslant p_1^x$ oraz $p_2^y \geqslant p_1^y$, to najwyższy pusty prostopadłościan, którego współrzędne x są nie większe od p_1^x ma wysokość p_2^z. Obliczamy objętość tego prostopadłościanu i ewentualnie uaktualniamy wartości zmiennych *maxobj* oraz *pkt*. Aktualizujemy także B — $B := B \cup \{p_2\}$. W przypadku $p_2^x \geqslant p_1^x$ oraz $p_2^y \leqslant p_1^y$ postępujemy podobnie.

Przypuśćmy, że rozważamy kolejny punkt p_{i+1}. Jeżeli istnieje taki punkt $p \in B$, że jednocześnie $p_{i+1}^x > p^x$ oraz $p_{i+1}^y > p^y$, to punkt p_{i+1} nie leży na ścianie żadnego pustego prostopadłościanu o większej objętości, a zatem nie wpływa na rozwiązanie. Przypuśćmy, że takiego punktu p nie ma. Zatem istnieją puste prostopadłościany o wysokości p_{i+1}^z, na których ścianach leżą punkty z B. Niech p_a będzie punktem z B takim, że $p_a^x < p_{i+1}^x$ oraz p_a ma największą współrzędną x spośród punktów w B o tej własności. Podobnie, niech p_b będzie odpowiednim punktem z B takim, że $p_b^y < p_{i+1}^y$ oraz p_b ma największą współrzędną y spośród punktów w B o tej własności. Te punkty znajdujemy przeglądając kolejno wszystkie elementy B. Rozważmy pary (s,t) kolejnych punktów z B począwszy od $s = p_a$, a kończąc na $t = p_b$, i odpowiednie puste prostopadłościany o wysokości p_{i+1}^z, na których ścianach leżą s i t. Obliczamy objętości wszystkich tych prostopadłościanów i jeżeli trzeba, uaktualniamy *maxobj* oraz *pkt*. Potem usuwamy te punkty z B, które są pomiędzy p_a i p_b, a na ich miejsce wstawiamy p_{i+1} (zob. rysunek 1). Jeżeli p_a lub p_b nie istnieje, to znaczy p_{i+1} jest krańcowym punktem w B — postępujemy podobnie.

Rys. 1 Ilustracja działania algorytmu przy rozpatrywaniu punktu $p = p_{i+1}$. Oś OZ jest prostopadła do płaszczyzny rysunku. Usuwane punkty brzegu zaznaczono symbolem $\circ$. Kolorem szarym zaznaczono podstawy rozpatrywanych prostopadłościanów (wysokości tych prostopadłościanów są równe p^z).



Po rozpatrzeniu ostatniego punktu p_n rozwiązanie będzie na zmiennej *pkt*. Asymptotyczny koszt algorytmu szacujemy przez $O(n^2)$, ponieważ rozpatrując kolejny punkt p_{i+1} potrzebujemy przejrzeć elementy w B w celu znalezienia punktów p_a oraz p_b, a rozmiar tego zbioru możemy oszacować ogólnie tylko przez n.

* Dla prostoty opisu zakładamy, że do zbioru A należy punkt $(1, 1, 10^6)$.

Rozwiązanie wzorcowe nie jest asymptotycznie najszybsze. Można je ulepszyć tak, by otrzymać pesymistyczny koszt $O(n \log n)$. W tym celu przechowujemy brzeg w wierzchołkach drzewa poszukiwań binarnych. Używamy drzewo zrównoważonego, które daje operacje wyszukiwania, usuwania i wstawiania z kosztem $O(\log n)$. Implementacja takich drzew jest czasochłonna, a uzyskane przyspieszenie nie jest znaczące dla rozmiaru danych określonych w specyfikacji zadania.

INNE ROZWIĄZANIA

Istnieją rozwiązania „naiwne", które opierają się na mniej systematycznym przeszukiwaniu zbioru wszystkich pustych prostopadłościanów.

Oto pierwsze z nich. Rozważamy wszystkie możliwe trójki punktów $T \subseteq A$. Każda z nich poprzez maksymalne wartości współrzędnych swoich elementów wyznacza regularny prostopadłościan, na którego ścianach leżą punkty z T. Sprawdzamy, czy ten prostopadłościan jest pusty przeglądając kolejno punkty z A. Podobnie, każda para punktów $P \subseteq A$ i każda współrzędna (na przykład x), wyznacza jednoznacznie prostopadłościan, którego krawędź x ma długość 10^6, a punkty z P leżą na jego ścianach. Sprawdzamy każdy z takich prostopadłościanów. Podobnie, każdy punkt $p \in A$, dla każdej pary współrzędnych (na przykład x i y), wyznacza jednoznacznie prostopadłościan, którego krawędzie x i y mają długość 10^6 i na którego ścianie leży p. Sprawdzamy wszystkie takie prostopadłościany. Złożoność tego algorytmu wynosi $O(n^4)$.

Oto podobne inne rozwiązanie. Rozważamy wszystkie pary punktów $\{p, q\} \subseteq A$. Ich rzuty na płaszczyznę XY wyznaczają boki prostokąta $R_{p,q}$, którego dwie krawędzie leżą na osiach X i Y a pozostałe zawierają punkty p i q. Przeglądamy wszystkie punkty z A i znajdujemy taki r, który ma minimalną wysokość spośród tych, których rzuty należą do $R_{p,q}$. Sprawdzamy pusty prostopadłościan, na którego ścianach leżą punkty p, q, r. Czas pracy takiego algorytmu wynosi $O(n^3)$.

TESTY

Do oceny rozwiązań zawodników użyto 9-ciu testów PUS0.IN–PUS8.IN. Test PUS0.IN był testem z treści zadania. Połowa z testów była wykonywana na małych zbiorach danych i sprawdzała poprawność rozwiązania. Pozostałe testy sprawdzały czas pracy rozwiązania na dużych danych losowych. Algorytm o koszcie $O(n^4)$ zaliczał test PUS5.IN, ale nie zaliczał testu PUS6.IN. Algorytm o koszcie $O(n^3)$ zaliczał test PUS6.IN, ale nie zaliczał testu PUS7.IN. Algorytm wzorcowy i algorytm o koszcie $O(n \log n)$ przechodziły wszystkie testy.