Zawody II stopnia

opracowania zadań

Marcin Sawicki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Bitmapa

Dana jest prostokątna bitmapa o rozmiarach $n \times m$. Każdy piksel bitmapy jest albo biały, albo czarny, przy czym co najmniej jeden jest biały. Piksel w i-tym wierszu i j-tej kolumnie bitmapy nazywamy pikselem (i, j). Odległość dwóch pikseli $p_1 = (i_1, j_1)$ oraz $p_2 = (i_2, j_2)$ określamy jako

$$d(p_1, p_2) = |i_1 - i_2| + |j_1 - j_2|.$$

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `BIT.IN` *opis bitmapy,*
- *dla każdego piksela oblicza odległość do najbliższego piksela białego,*
- *zapisuje wyniki w pliku tekstowym* `BIT.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `BIT.IN` *znajduje się para liczb całkowitych n, m oddzielonych pojedynczym odstępem, $1 \leqslant n \leqslant 182$, $1 \leqslant m \leqslant 182$.*

W każdym z kolejnych n wierszy pliku `BIT.IN` *zapisano dokładnie jedno słowo zero-jedynkowe o długości m — opis jednego wiersza bitmapy. Na pozycji j w wierszu $(i + 1)$, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, znajduje się '1' wtedy i tylko wtedy, gdy piksel (i, j) jest biały.*

Wyjście

W i-tym wierszu pliku wyjściowego `BIT.OUT`, *$1 \leqslant i \leqslant n$, należy wypisać m liczb całkowitych $f(i, 1), \ldots, f(i, m)$ pooddzielanych pojedynczymi odstępami i takich, że $f(i, j)$ jest odległością piksela (i, j) od najbliższego piksela białego.*

Przykład

Dla pliku wejściowego `BIT.IN`*:*

```
3 4
0001
0011
0110
```

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

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

ROZWIĄZANIE

Zacznijmy od bardziej abstrakcyjnego sformułowania problemu. Niech X będzie dowolnym zbiorem. Jego elementy będą dla nas *punktami*, dlatego też cały X będziemy nazywać *przestrzenią*.

Definicja 1: *Metryka* lub inaczej *odległość* w przestrzeni X jest to funkcja dwu zmiennych $d : X \times X \longrightarrow [0; +\infty)$, która dla dowolnych punktów p, q, r spełnia następujące warunki:

(1) $d(p,q) = 0 \iff p = q$

(2) $d(p,q) = d(q,p)$

(3) $d(p,q) + d(q,r) \geqslant d(p,r)$

Pierwsze dwa z tych warunków są dość oczywiste, a trzeci, nazywany *nierównością trójkąta*, oznacza, że gdy idziemy z p do r przez jakiś pośredni punkt q, to nie możemy nic zaoszczędzić w stosunku do bezpośredniej drogi z p do r.

Znanym ze szkoły przykładem metryki jest *odległość euklidesowa* na płaszczyźnie określona dla punktów $p = (x_1, y_1)$ oraz $q = (x_2, y_2)$ wzorem

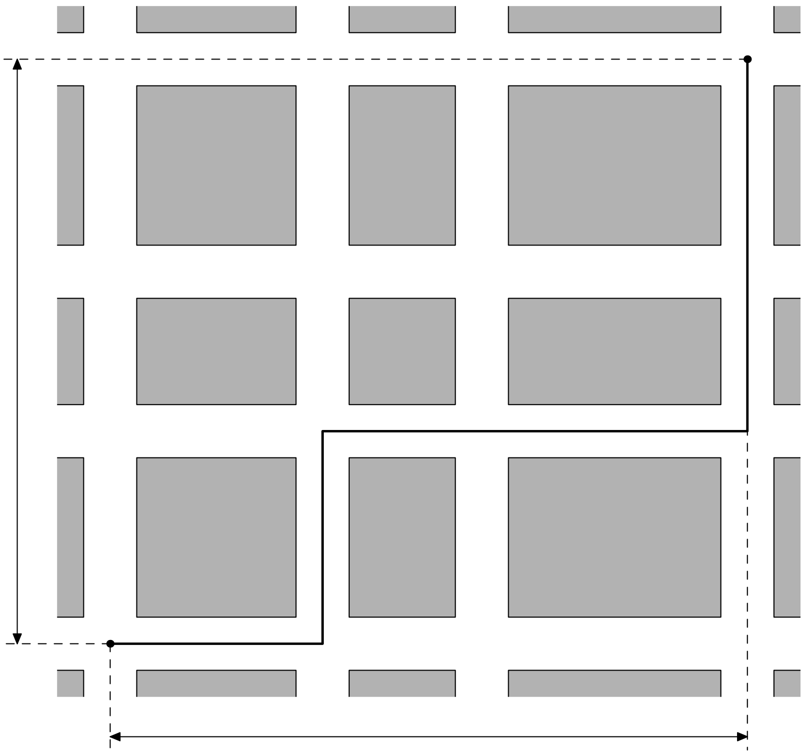
$$d_e(p,q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Także podany w treści zadania wzór określa metrykę, którą nazywa się *metryką miejską* lub *taksówkarską*:

$$d_m(p,q) = |x_1 - x_2| + |y_1 - y_2|$$

Skąd taka nazwa? Wyobraźmy sobie miasto, w którym ulice biegną tylko w dwu wzajemnnie prostopadłych kierunkach. Jaką odległość musi ponokać samochód, by z jednego skrzyżowania dostać się na inne? Odpowiedź znajdziesz na rysunku 1.

Rys. 1 Metryka miejska.



Jeśli mamy określoną metrykę, możemy też określić *odległość punktu $p \in X$ od zbioru $A \subset X$* wzorem

$$d(p,A) = \inf_{q \in A} d(p,q).$$

Stosujemy to samo co poprzednio oznaczenie $d(\cdot,\cdot)$. Oczywiście nie prowadzi to do nieporozumień, bo $d(p,q) = d(p,\{q\})$.

Nasze zadanie można więc sformułować tak: dla skończonej przestrzeni $X = \{1, 2, \ldots, n\} \times \{1, 2, \ldots, m\}$ z metryką miejską d_m oraz zadanego zbioru punktów białych $B \subset X$ należy, wyznaczyć każdą z odległości $d_m(x, B)$, dla $x \in X$.

Rozwiązanie wzorcowe wykorzystuje specyficzną własność metryki miejskiej w naszej przestrzeni. Jeśli rozpatrzymy graf $G = (X, E)$, którego wierzchołkami będą punkty zbioru X, a krawędzie będą łączyły każdy punkt z jego bezpośrednimi sąsiadami (czterema, chyba że leży on na brzegu bitmapy), to odległość dowolnych dwu punktów $d_m(p, q)$ równa jest ich odległości w rozważanym grafie, to znaczy długości najkrótszej łączącej je ścieżki. Fakt ten jest intuicyjnie oczywisty, a podanie formalnego dowodu może być dla Czytelnika pouczającym ćwiczeniem (dla uproszczenia proponuję rozważać nieskończony zbiór $X = \mathbf{Z} \times \mathbf{Z}$, gdzie $\mathbf{Z}$ oznacza zbiór wszystkich liczb całkowitych).

Do znajdowania najkrótszych ścieżek w grafie możemy z powodzeniem zastosować przeszukiwanie wszerz (BFS). Kolejkę zainicjujemy jednak nie jednym punktem, lecz od razu wszystkimi punktami białymi. Dla każdego z tych punktów szukana odległość d_m wynosi oczywiście 0. Natomiast dla dowolnego innego punktu jest ona o 1 większa, niż odległość jego poprzednika w drzewie (lub raczej lesie) BFS.

Aby udowodnić poprawność naszego algorytmu wystarczy nieznacznie zmodyfikować dowód znanego faktu, że zwyczajny BFS (startujący z jednego punktu) poprawnie oblicza odległości od tego punktu. Zainteresowanych odsyłam do znakomitej książki [12], do rozdziału 23.2 i zawartego w nim twierdzenia 23.4.

Złożoność czasowa naszego algorytmu jest taka sama, jak zwykłego BFS, czyli $O(|V| + |E|)$. U nas $|V| = |X| = nm$, zaś $|E| \leqslant 2|X|$, zatem wynosi ona $O(nm)$. Jest więc liniowa ze względu na rozmiar danych.

INNE ROZWIĄZANIA

Naiwne rozwiązanie mogłoby polegać na znalezieniu dla każdego punktu p z osobna najbliższego mu punktu białego. Można by użyć algorytmu BFS lub po prostu przeglądać kolejno zbiory tych punktów, których odległość d_m od p wynosi odpowiednio $1, 2, 3, \ldots$ Łatwo wyznaczyć te zbiory. Są to brzegi kwadratów o bokach nachylonych pod kątem 45° do osi układu współrzędnych i środkach w punkcie p. Można te zbiory traktować jako okręgi w metryce d_m o środku w punkcie p. Oznaczmy te zbiory przez $C_m(p, R)$ ($R \geqslant 0$ — promień). Koszt takiego rozwiązania w pesymistycznym przypadku wyniósłby $\Omega(n^2 m^2)$ (np. dla jednego białego punktu w rogu planszy).

Istotne ulepszenie powyższego algorytmu opiera się na spostrzeżeniu, że z nierówności trójkąta dla dowolnej metryki d łatwo wynika nierówność

$$|d(p, A) - d(q, A)| \leqslant d(p, q)$$

Oznacza to, że można wykorzystać ideę programowania dynamicznego: jeżeli p i q są sąsiadami, a zatem $d(p, q) = 1$, to do obliczenia $d(q, B)$ możemy wykorzystać wartość $d(p, B)$. Wiemy, że $d(q, B)$ może różnić się od $d(p, B)$ o jeden lub zero. Wystarczy zatem w poszukiwaniu punktów białych przejrzeć najpierw zbiór $C_m(q, R-1)$, a następnie $C_m(q, R)$, gdzie $R = d(p, B)$. Obniża to koszt naszego rozwiązania do $O(nm \min(n, m))$, gdyż jak nietrudno spostrzec, $|C_m(p, R)| \leqslant 2 \min(n, m)$ (nawet dla $R > \min(n, m)$ nasze kwadratowe „okręgi" są zbiorami skończonymi!).

Ostatni z przedstawionych pomysłów jest o tyle ciekawy, że pozwala rozwiązać zadanie także w przypadku metryki euklidesowej d_e, a taka była oryginalna wersja prezentowanego zadania (stąd ograniczenie $1 \leqslant n, m \leqslant 182$ — wtedy kwadrat odległości naroży bitmapy spełnia $d_e^2 = 2 \cdot 181^2 = 65522 < 2^{16}$, czyli mieści się w jednym słowie maszynowym). Przypuśćmy mianowicie, że znamy odległość $R = d_e(p, B)$, a q jest sąsiadem p. Zatem $d_e(p, q) = 1$ i dla wyznaczenia $d_e(q, B)$ wystarczy w poszukiwaniu punktów ze zbioru B przejrzeć te punkty $x \in X$, dla których $R - 1 \leqslant d_e(x, q) < R + 1$ (warunek ten wyznacza na płaszczyźnie pierścień kołowy).

Jak wyznaczyć te punkty efektywnie? Posortujmy punkty przestrzeni X po wielkości $d_e(x, p_0)$, gdzie $p_0 = (1, 1)$ (w praktyce do porównań będziemy używać liczby d_e^2, która jest całkowita). Teraz łatwo znajdziemy te z nich, których odległości od p_0 spełniają określoną nierówność, np. przy pomocy wyszukiwania binarnego.

Przesuwając te punkty o wektor $q - p_0$, uzyskamy jedną z ćwiartek szukanego pierścienia. Pozostałe jego punkty wyznaczymy przez symetrie. Można by poczynić dodatkowe oszczędności zauważając, że taki pierścień kołowy na siatce punktów o współrzędnych całkowitych ma nie tylko symetrię czworokątną, lecz nawet ośmiokątną (cztery osie symetrii).

I tym razem widzimy, że punktów spełniających warunek $R - 1 \leqslant d_e(x, q) < R + 1$ jest na pewno nie więcej niż $K \cdot \min(m, n)$ (proste, choć nieco żmudne rozważania geometryczne wykazują, że można przyjąć np. $K = 8$). W takim razie koszt pesymistyczny głównej części naszego algorytmu nie zmieni się w stosunku do algorytmu dla metryki miejskiej i wyniesie $O(nm \min(n, m))$. Dodatkowy narzut czasowy na posortowanie punktów X na samym początku wynosi $\Theta(nm(\log n + \log m)) = O(nm \min(n, m))$, o ile przyjmiemy $\log n = O(m)$ i $\log m = O(n)$. Możemy zatem powiedzieć, że wszystkie obliczenia wykonamy w czasie $O(nm \min(n, m))$.

Inspiracją dla tego zadania był następujący problem: jak szybko projektować efektowne stereogramy? Do wygenerowania stereogramu potrzebna jest mapa głębi poszczególnych pikseli. W najprostszym przypadku jest ona dwu- lub kilkupoziomowa, jednak utworzone na tej podstawie stereogramy są mało ciekawe. Z drugiej strony, utworzenie mapy głębi na podstawie projektu przestrzennego (np. metodą śledzenia promieni) może być bardzo pracochłonne.

Dobrym pomysłem może być tworzenie stereogramu na podstawie czarno-białego rysunku, na którym białe plamy oznaczają obszary płaskie (morze), natomiast czarne — wypukłości (lądy). Wysokość nad poziomem morza zmienia się płynnie z odległością od morza (niekoniecznie liniowo). Utworzenie mapy głębi na podstawie projektu wymaga właśnie rozwiązania naszego zadania.

Pomysł ten zaprogramowałem w javie. Działający applet można znaleźć na stronie internetowej Olimpiady Informatycznej.

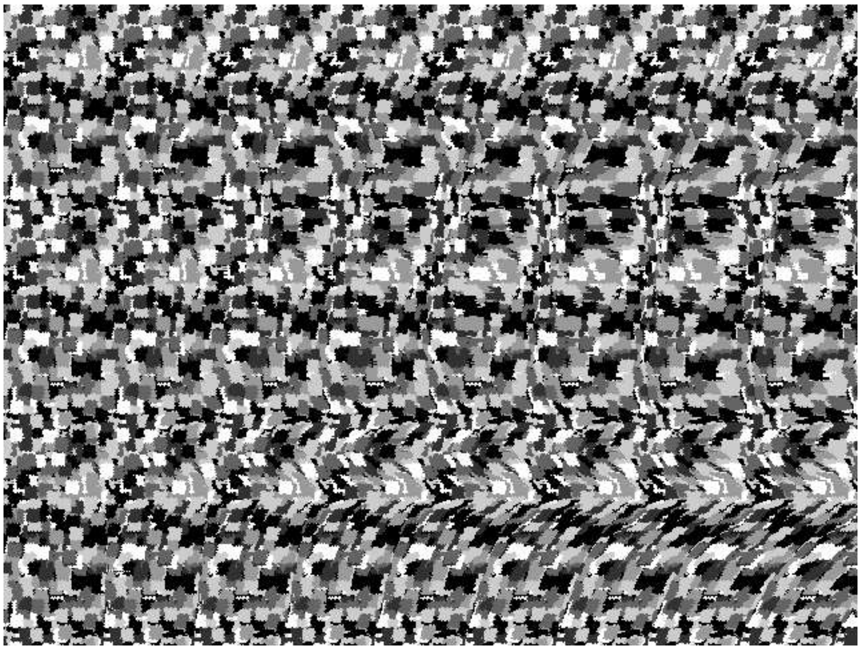
TESTY

Rozwiązania zawodników były testowane przy pomocy 16 testów BIT0.IN–BIT15.IN. Test BIT0.IN był testem z treści zadania. Kolejne cztery testy to bitmapy 1×1, 182×1, 1×182 i 182×182. Każda z nich miała po jednym białym pikselu. Następne osiem testów stanowiły bitmapy maksymalnego rozmiaru, składające się z czarnego pola, otoczonego wąską białą nieregularną obwódką. Są to testy wydajnościowe dla algorytmu opisanego w dyskusji, gdyż koszt wykonania jednego kroku jest tam proporcjonalny do obliczanej odległości, a zatem duży dla dużych czarnych obszarów. Nieregularny kształt obwódki był zbyt trudny dla prostych algorytmów heurystycznych. Ostatnie trzy testy miały również największy możliwy rozmiar i składały się z białych napisów lub prostych rysunków na czarnym tle.

Rys. 2 Projekt stereogramu

Rys. 3 Mapa głębi wygenerowana na podstawie projektu z rysunku 2

Rys. 4 Otrzymany stereogram



Marcin Kubica
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Grotołazi

Drużyna grotołazów organizuje trening w największej jaskini Bajtogór. Trening polega na przebyciu drogi od komory leżącej najwyżej, do komory leżącej najniżej. Grotołazi mogą posuwać się tylko w dół, tzn. kolejne odwiedzane komory muszą leżeć coraz niżej. Dodatkowo każdy z nich ma wyjść z najwyższej komory innym korytarzem oraz każdy z nich ma wejść do najniższej komory innym korytarzem. Pozostałe korytarze grotołazi mogą pokonywać wspólnie. Ilu grotołazów może odbyć trening jednocześnie?

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `GRO.IN` *opis jaskini,*
- *oblicza największą liczbę grotołazów, którzy mogą odbyć trening jednocześnie,*
- *zapisuje wynik w pliku tekstowym* `GRO.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `GRO.IN` *jest zapisana jedna liczba całkowita n ($2 \leqslant n \leqslant 200$), równa liczbie komór w jaskini. Komory są ponumerowane liczbami od 1 do n w taki sposób, że komora z większym numerem leży niżej od komory z numerem mniejszym. (Najwyższa komora ma numer 1, a najniższa n.) W wierszach o numerach $2, 3, \ldots, n$ są opisane korytarze wychodzące z kolejnych komór $1, 2, \ldots, n-1$, do komór o wyższych numerach (w wierszu o numerze i znajduje się opis korytarzy wychodzących z komory o numerze $i-1$). Każdy z tych wierszy zawiera ciąg nieujemnych liczb całkowitych pooddzielanych pojedynczymi odstępami. Pierwsza liczba w wierszu, m, $0 \leqslant m \leqslant (n-i+1)$, jest liczbą korytarzy prowadzących z komory do komór o wyższych numerach, natomiast kolejne m liczb to numery komór, do których te korytarze prowadzą.*

Wyjście

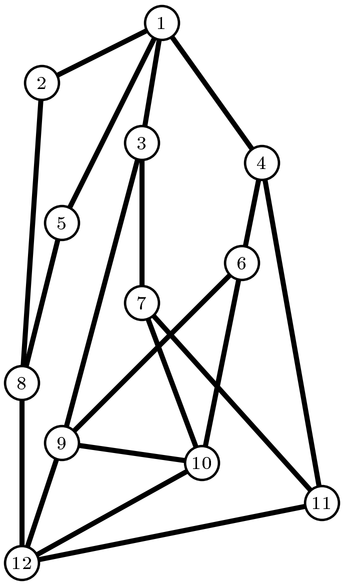
W jedynym wierszu pliku `GRO.OUT` *Twój program powinien zapisać jedną liczbę całkowitą, równą maksymalnej liczbie grotołazów mogących wziąć jednocześnie udział w treningu.*

Przykład

Dla pliku wejściowego `GRO.IN`*:*

```
12
4 3 4 2 5
1 8
2 9 7
2 6 11
1 8
2 9 10
2 10 11
1 12
2 10 12
1 12
1 12
```

opisującego następującą jaskinię:



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

```
3
```

ROZWIĄZANIE

Warunek nałożony w treści zadania na drogi grotołazów mówi, że mają oni wyjść z najwyższej komory różnymi korytarzami i wejść do najniższej komory różnymi korytarzami. Pomiędzy rozejściem się z najwyższej komory i spotkaniem w najniższej komorze ich drogi mogą przebiegać dowolnie. Nazwijmy komory połączone korytarzami bezpośrednio z najwyższą komorą komorami górnymi, a komory połączone korytarzami bezpośrednio z najniższą komorą komorami dolnymi. (Przyjmujemy przy tym, że komora najwyższa nie jest dolną, a komora najniższa nie jest górną, nawet jeśli są połączone bezpośrednim korytarzem.) Każda droga wiodąca z najwyższej do najniższej komory wiedzie albo korytarzem łączącym bezpośrednio najwyższą i najniższą komorę, albo poprzez pewną komorę górną oraz pewną komorę dolną. Zauważmy, że wynik zależy nie tyle od przebiegu korytarzy łączących komory górne z dolnymi, co od tego, które pary komór dolnych i górnych są połączone drogami. Pozwala nam to podzielić obliczenie wyniku na dwie fazy. Najpierw wyznaczamy komory górne i dolne oraz ustalamy pary komór (górna, dolna) połączone drogami. Następnie obliczamy wynik na podstawie rezultatów pierwszej fazy algorytmu (uwzględniając ewentualny korytarz łączący bezpośrednio najwyższą i najniższą komorę).

Spójrzmy na komory i korytarze jak na wierzchołki i krawędzie grafu skierowanego. Pierwsza faza algorytmu sprowadza się do obliczenia domknięcia zwrotno-przechodniego grafu. (Domknięcie takie jest grafem, który zawiera dokładnie takie krawędzie $v \to w$, że $v = w$, lub z v można dojść do w.) Rozmiar danych ($2 \leqslant n \leqslant 200$) pozwala nam na przechowywanie grafu w postaci tablicy sąsiedztwa.*

```
const MAXN = 200;
var
   T : array [1..MAXN, 1..MAXN] of boolean;
   n : word;
```

Domknięcie zwrotno-przechodnie grafu możemy obliczyć korzystając z wersji algorytmu Floyda-Warshalla (por. [12], p. 26.2, [12], p. 3.5). Dużą zaletą tego algorytmu jest prostota jego implementacji.

```
var
```

* Fragmenty programu przytaczane w tym rozdziale pochodzą z rozwiązania zamieszczonego na załączonej dyskietce.

```
  k, i, j : word;
begin
  for k := 1 to n do
    for i := 1 to n do
      if T[i, k] then
        for j := 1 to n do
          T[i, j] := T[i, j] or T[k, j];
  for i := 1 to n do T[i, i] := true
end
```

W dalszej części interesować nas będzie tylko to, czy z danej komory górnej można dojść do danej komory dolnej. Zbiory komór górnych i dolnych możemy pamiętać w postaci tablic logicznych o wartościach równych **true** dla indeksów należących do zbioru. Zbiory te możemy wyznaczyć następująco:

```
var
  A, B : array [1..MAXN] of boolean;
  i : word;
begin
  for i := 1 to n do begin
    A[i] := T[1, i];
    B[i] := T[i, n]
  end
end
```

Wyobraźmy sobie teraz graf dwudzielny (nieskierowany), którego wierzchołkami są komory górne i dolne, a krawędzie są postaci $v - w$, dla takich komór górnych v i komór dolnych w, że z v można dojść do w. Przez każdą górną i dolną komorę może przejść co najwyżej jeden grotołaz. Tak więc szukana liczba grotołazów jest maksymalnym rozmiarem skojarzenia w naszym grafie dwudzielnym. (Skojarzenie to taki zbiór $\{v_1 - w_1, \ldots, v_k - w_k\}$ krawędzi grafu dwudzielnego, że $v_1, \ldots, v_k, w_1, \ldots, w_k$ są parami różne.) Algorytm znajdowania skojarzenia o maksymalnej liczności w grafie dwudzielnym jest opisany np. w [18], p. 4.3, oraz w [12], p. 27.3.

Potrzebne nam będzie pojęcie *ścieżki rozszerzającej*. Niech S będzie pewnym skojarzeniem. *Ścieżką rozszerzającą* skojarzenie S nazwiemy każdą taką ścieżkę prostą $v_1 - v_2 - \cdots - v_{2l}$ (dla $1 \leqslant l$), że: v_1 i v_{2l} nie są końcami żadnej krawędzi z S, $v_2 - v_3, \ldots, v_{2l-2} - v_{2l-1} \in S$, oraz $v_1 - v_2, \ldots, v_{2l-1} - v_{2l} \notin S$. Inaczej mówiąc, ścieżka rozszerzająca to taka ścieżka prosta łącząca wierzchołki nie pojawiające się w skojarzeniu, której krawędzie na przemian nie należą i należą do skojarzenia. Zauważmy, że

$$S' = S \setminus \{v_2 - v_3, \ldots, v_{2l-2} - v_{2l-1} \in S\} \cup \{v_1 - v_2, \ldots, v_{2l-1} - v_{2l} \notin S\}$$

jest też skojarzeniem i to mającym o jeden element więcej niż S. Okazuje się, że zawsze, gdy skojarzenie nie ma maksymalnej liczności, to istnieje względem niego ścieżka rozszerzająca (por. [18], p. 4.3).

Zauważmy, że pusty zbiór krawędzi jest skojarzeniem. Nasz algorytm polega na ciągłym rozszerzaniu (początkowo pustego) skojarzenia, aż do osiągnięcia skojarzenia o maksymalnej liczności. Do rozwiązania pozostaje jeszcze tylko jeden problem — jak znaleźć ścieżkę rozszerzającą. W zależności od skonstruowanego do tej pory skojarzenia S skierujemy nasz graf dwudzielny w następujący sposób. Niech $v - w$ będzie krawędzią, gdzie v jest komorą górną, a w komorą dolną. Jeżeli $v - w \in S$, to $v \leftarrow w$, a w przeciwnym przypadku $v \rightarrow w$. Zauważmy, że w tak zorientowanym grafie, ścieżki rozszerzające są dokładnie ścieżkami prowadzącymi od pewnej komory górnej, nie występującej w skojarzeniu, do pewnej komory dolnej, także nie występującej w skojarzeniu. Możemy więc szukać takich ścieżek np. za pomocą przeszukiwania w głąb.

Konstruowane skojarzenie możemy reprezentować za pomocą dwóch tablic zawierających dla każdej komory górnej numer skojarzonej z nią komory dolnej i dla każdej komory dolnej numer skojarzonej z nią komory górnej (lub 0, gdy dana komora nie należy do skojarzenia).

```
  var SA, SB : array [1..MAXN] of word;
```

Poniższa funkcja szuka ścieżki rozszerzającej zaczynającej się w komorze o numerze zadanym jako argument (zakładamy, że jest to komora górna nie występująca w skojarzeniu). Korzysta ona z pomocniczej tablicy *OB* służącej do zaznaczania odwiedzonych już wierzchołków. Jej wynikiem jest **true**, gdy taką ścieżkę udało się znaleźć. Równocześnie funkcja ta rozszerza dotychczasowe skojarzenie.

```
var
  OB : array [1..MAXN] of boolean;
function Powieksz(i : word) : boolean;
var j : word;
begin
  for j := i to n do
    if T[i, j] and B[j] and not OB[j] and (SA[i] ≠ j) then begin
      { Krawędź i–j należy do grafu, ale nie do skojarzenia,
        j jest nieodwiedzoną jeszcze komorą dolną. }
      OB[j] := true;
      if (SB[j] = 0) or Powieksz(SB[j]) then begin
        { Ścieżka znaleziona }
        SA[i] := j;
        SB[j] := i;
        Powieksz := true;
        Exit
      end
    end;
  Powieksz := false
end;
```

Zwróć uwagę, że funkcja ta korzysta z „leniwego” obliczania warunków logicznych.

Niech *ns* będzie zmienną przechowującą liczność konstruowanego skojarzenia. Następująca procedura konstruuje skojarzenie o maksymalnej liczności. Na zakończenie, wyliczając wynik na zmiennej *ns*, uwzględnia ona również przypadek, gdy najwyższa i najniższa komora są połączone bezpośrednim korytarzem.

```
var
  ns : word;
procedure Skojarz;
var
  i, j : word;
  pow : boolean;
begin
  ns := 0;
  for i := 1 to n do begin
    SA[i] := 0; SB[i] := 0
  end;
  for j := 1 to n do OB[j] := false;
  repeat
    pow := false;
    for i := 1 to n do
      if A[i] and (SA[i] = 0) then begin
        { Szukamy ścieżki rozszerzającej zaczynającej się w i }
        if Powieksz(i) then begin
          pow := true;
          Inc(ns);
          for j := 1 to n do OB[j] := false
        end
      end
  until not pow;
  if T[1, n] then Inc(ns)
end;
```

Jak łatwo zauważyć, pierwsza faza algorytmu działa w czasie $O(n^3)$. Znalezienie ścieżki rozszerzającej wymaga co najwyżej tyle czasu, co przeszukanie całego grafu, czyli $O(n^2)$. Ścieżek rozszerzających może być znalezionych co najwyżej $n-2$. Tak więc cały algorytm działa w czasie $O(n^3)$.

Należy nadmienić, że opisany algorytm obliczania najliczniejszego skojarzenia nie jest optymalny. Znany jest inny, bardziej skomplikowany algorytm (por. [18], p. 4.3) o czasie działania $O(n^{5/2})$. Jednak przy takim rozmiarze danych jak w zadaniu, taka różnica nie jest istotna, gdyż dominującą częścią programu jest wczytywanie danych z dysku.

Zastanówmy się nad innymi możliwymi rozwiązaniami tego zadania. Możemy je podzielić na takie, w których wyliczane jest domknięcie zwrotno-przechodnie grafu komór i korytarzy, oraz na takie, w których konstruowane są całe drogi grotołazów.

Domknięcie zwrotno-przechodnie grafu możemy wyliczać inaczej niż za pomocą algorytmu Warshalla. Możemy dla każdej komory górnej wyznaczać komory dolne, do których możemy dojść np. za pomocą przeszukiwania w głąb. Nie wpływa to w istotny sposób na czas działania programu.

Jak inaczej możemy znaleźć maksymalne skojarzenie? Jedna z możliwości, to rekurencyjne przeglądanie z nawrotami wszystkich możliwych skojarzeń. Algorytm taki daje poprawne wyniki i stosunkowo łatwo go zaimplementować. Niestety czas jego działania jest wykładniczy. Tak więc czas obliczeń jest akceptowalny tylko dla niewielkich testów.

Zastanówmy się nad programami, które starają się zbudować skojarzenie o maksymalnej liczności w sposób zachłanny. Możliwych jest kilka sposobów wyboru kolejnej krawędzi skojarzenia. Można wybierać „pierwszą wolną" krawędź, losową, lub wybierać krawędź o końcach będących wierzchołkami jak najmniejszego stopnia. Ta ostatnia technika daje najlepsze rezultaty. Niestety zawodzi dla większych testów losowych oraz dla testów odpowiednio „złośliwych". W przypadku losowego wyboru krawędzi możemy wielokrotnie konstruować losowe skojarzenie i wybierać najbardziej liczne. Taka technika polepsza rezultaty tylko w przypadku niewielkich testów.

Algorytmy konstruujące drogi grotołazów, bez wyliczenia domknięcia grafu, dają generalnie gorsze rezultaty. Możemy rekurencyjnie (z nawrotami) poszukiwać maksymalnej liczby dróg grotołazów. Algorytm taki ma koszt wykładniczy i, jak poprzednio, ma szansę działać w rozsądnym czasie tylko dla małych testów. Stosowanie heurystyk mających na celu ograniczenie liczby przeszukiwanych dróg nie zmienia w istotny sposób sytuacji.

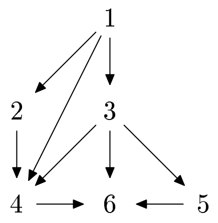
Reasumując, rozwiązania nie wyliczające domknięcia zwrotno-przechodniego grafu i nie korzystające ze ścieżek rozszerzających miały szansę na zaliczenie tylko części testów — zwykle niewielkich.

TESTY

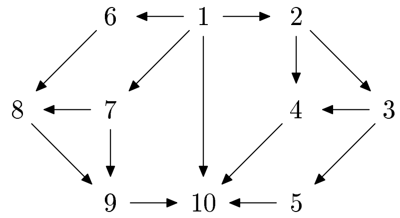
Do sprawdzenia rozwiązań zawodników użyto 11-tu testów GRO0.IN–GRO10.IN. Test GRO0.IN był testem z treści zadania. Testy GRO1.IN–GRO4.IN są niewielkimi testami poprawnościowymi (zob. ilustracje).

Testy GRO5.IN–GRO8.IN są testami losowymi różnej wielkości, przy czym testy GRO5.IN i GRO8.IN, z pominięciem najwyższej i najniższej komory, to grafy dwudzielne. Test GRO9.IN to duży test eliminujący rozwiązania zachłanne. Test GRO10.IN to graf o 200 komorach i wszystkich możliwych korytarzach. Poniższa tabelka przedstawia liczbową charakterystykę testów. Wszystkie testy można znaleźć na załączonej dyskietce.

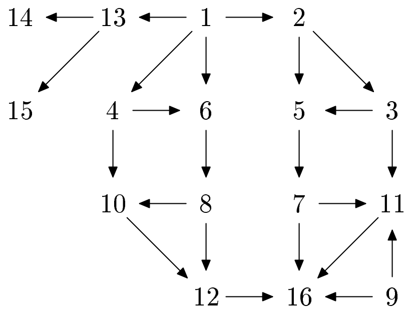
nr testu	l. komór	l. korytarzy	wynik
1	6	9	2
2	10	15	3
3	16	23	2
4	18	43	8
5	100	248	48
6	200	400	43
7	200	498	85
8	200	498	95
9	198	628	98
10	200	19900	199



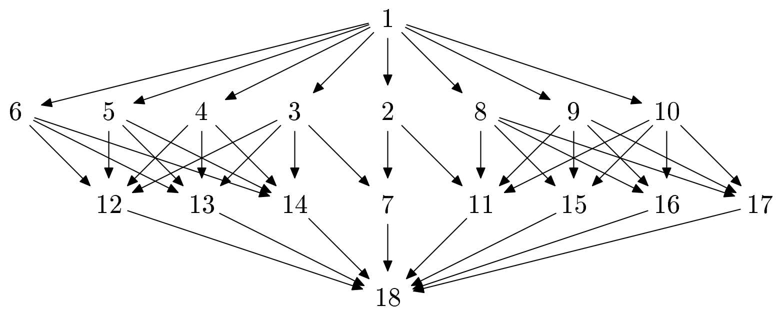
GRO1.IN



GRO2.IN



GRO3.IN



GRO4.IN

Wojciech Rytter
autor zadania, opracowanie

Adam Borowski
program wzorcowy

Lunatyk

Płaski dach budynku ma kształt kwadratu o rozmiarach $3^k \times 3^k$ i bokach równoległych do kierunków północ-południe oraz wschód-zachód. Dach pokryto kwadratowymi kaflami o boku 1, ale jeden kafel został wyrwany i w tym miejscu jest dziura. Kafle na dachu tworzą prostokątną siatkę, wobec tego ich pozycje można określić za pomocą współrzędnych. Kafel położony w południowo-zachodnim rogu ma współrzędne $(1, 1)$. Pierwsza współrzędna rośnie przy przemieszczaniu się na wschód, a druga przy przemieszczaniu się na północ.

Lunatyk przemierza dach przemieszczając się w każdym kroku z kafla, na którym aktualnie się znajduje, na kafel sąsiedni od wschodu (E), zachodu (W), południa (S) lub północy (N). Wędrówka lunatyka po dachu zaczyna się zawsze od kafla w rogu południowo-zachodnim, a opisem drogi jest słowo d_k złożone z liter N, S, E, W oznaczających, odpowiednio, krok na północ, na południe, na wschód i na zachód. Dla $k = 1$ opisem drogi lunatyka jest słowo

$$d_1 = EENNWSWN.$$

Dla $k = 2$ opisem drogi lunatyka jest słowo

$$\begin{aligned} d_2 = &NNEESWSEENNEESWSEEEENNWSWNNEENNWSW\text{-}\\ &NNEENNWSWNWWWSSENESSSSWWNENWWSSW\text{-}\\ &WNENWNEENNWSWN. \end{aligned}$$

Spójrz na rysunek przedstawiający drogi lunatyka po dachach o rozmiarach 3×3 i 9×9.

Ogólnie, dla dowolnego $k \geqslant 1$, opisem drogi lunatyka po dachu o rozmiarach $3^{k+1} \times 3^{k+1}$ jest słowo:

$$d_{k+1} = a(d_k)\ E\ a(d_k)\ E\ d_k\ N\ d_k\ N\ d_k\ W\ c(d_k)\ S\ b(d_k)\ W\ b(d_k)\ N\ d_k$$

gdzie funkcje a, b i c oznaczają następujące przemianowania liter określających kierunki:

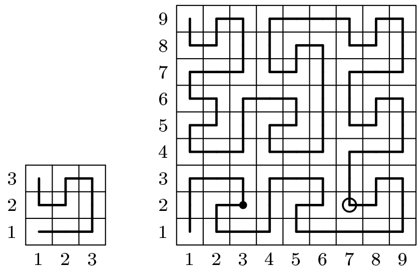
$$\begin{array}{llll} a: & E \to N & W \to S & N \to E & S \to W \\ b: & E \to S & W \to N & N \to W & S \to E \\ c: & E \to W & W \to E & N \to S & S \to N \end{array}$$

np. $a(SEN) = WNE$, $b(SEN) = ESW$, $c(SEN) = NWS$.

Zaczynamy obserwować lunatyka w momencie, gdy znajduje się na kaflu o współrzędnych (u_1, u_2). Po ilu krokach lunatyk wpadnie do dziury, która znajduje się w kwadracie o współrzędnych (v_1, v_2)?

Przykład

Na rysunku przedstawiono drogi lunatyka po dachu o rozmiarach 3×3 oraz po dachu o rozmiarach 9×9. W drugim przypadku zaznaczono punkt, od którego rozpoczynamy obserwację lunatyka i dziurę w dachu. Lunatyka dzieli od dziury 20 kroków.



Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `LUN.IN` *liczbę naturalną k określającą rozmiary dachu ($3^k \times 3^k$), pozycję lunatyka w chwili, gdy rozpoczynamy jego obserwację oraz pozycję dziury,*
- *oblicza liczbę kroków, które dzielą lunatyka od dziury,*
- *zapisuje wynik w pliku tekstowym* `LUN.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `LUN.IN` *jest zapisana jedna liczba naturalna k, $1 \leqslant k \leqslant 60$, określająca rozmiary dachu ($3^k \times 3^k$).*

W każdym z kolejnych dwóch wierszy pliku `LUN.IN` *są zapisane dwie liczby naturalne x, y oddzielone odstępem, $1 \leqslant x \leqslant 3^k$, $1 \leqslant y \leqslant 3^k$. Liczby w drugim wierszu pliku* `LUN.IN` *są współrzędnymi kafla, na którym stoi lunatyk; liczby w trzecim wierszu pliku* `LUN.IN` *są współrzędnymi dziury.*

Możesz założyć, że dane są tak dobrane, iż po pewnej liczbie kroków lunatyk wpadnie do dziury.

Wyjście

Jedyny wiersz pliku wyjściowego `LUN.OUT` *powinien zawierać liczbę kroków, które dzielą lunatyka od dziury.*

Przykład

Dla pliku wejściowego `LUN.IN`*:*

```
2
3 2
7 2
```

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

```
20
```

ROZWIĄZANIE

Zadanie sprowadza się do znalezienia odległości $DIST(k, x, y)$ od punktu (1,1) do zadanego punktu (x, y) w kwadracie $3^k \times 3^k$. Jeśli policzymy tę odległość dla punktu, od którego rozpoczynamy obserwację lunatyka i punktu położenia dziury, to wystarczy policzyć ich różnicę, aby otrzymać wynik końcowy.

Uwaga. W celu uproszczenia opisu rozwiązania przyjmujemy, że współrzędne są liczone od 0, a nie od 1, jak to jest w treści zadania.

Zadanie ma charakter rekurencyjny. Najistotniejszą cechą jest to, że mamy do czynienia z bardzo dużymi kwadratami i wędrowanie po takich kwadratach „małymi krokami” wymaga zbyt dużego czasu.

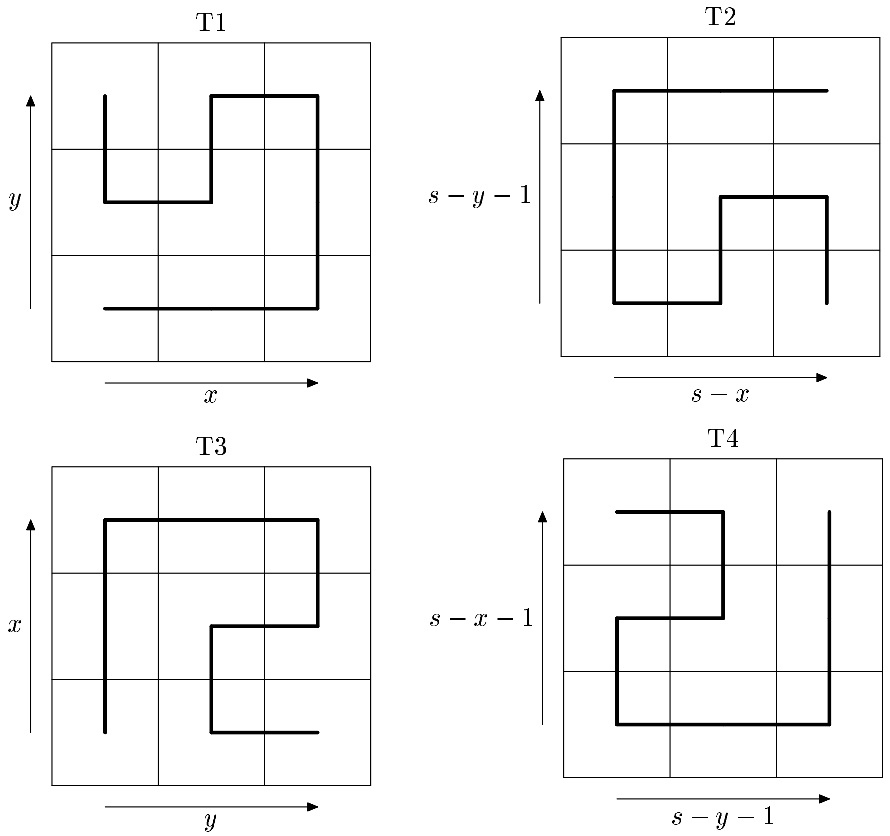
Możemy zaoszczędzić czas „przeskakując” przez całe duże podkwadraty, o ile wiemy, że lunatyk przejdzie w nich wszystkie pola. Tego typu przeskoki są pokazane na rysunku 3.

Kwadrat o rozmiarach $3^k \times 3^k$ nazwiemy kwadratem stopnia k. Składa się on z 9 podkwadratów stopnia $k-1$, które nazwiemy podkwadratami bazowymi. Każdy taki podkwadrat odpowiada parze (i, j) w naturalny sposób, gdzie $0 \leqslant i, j \leqslant 2$. Oznaczmy go przez $SUB(i, j)$. W każdym z tych bazowych podkwadratów lunatyk chodzi tak samo, z dokładnością do odpowiedniej transformacji współrzędnych. Transformacje te są podane na rysunku 1 i zapamiętane w tablicy TR. W tablicy NUM pamiętamy, w jakiej kolejności lunatyk odwiedza podkwadraty bazowe (zob. rysunek 2). Jeśli punkt (x, y) jest w podkwadracie bazowym o współrzędnych (baz_x, baz_y), to lunatyk chodzi tam zmieniając współrzędne zgodnie z $TR(baz\_x, baz\_y)$. Wcześniej możemy przeskoczyć $NUM(baz\_x, baz\_y)$ podkwadratów stopnia $k-1$.

Istnieje prosty sposób na policzenie współrzędnych (baz_x, baz_y) bazowego podkwadratu, w którym jest punkt (x, y), oraz współrzędnych punktu (wew_x, wew_y), będących współrzędnymi punktu (x, y) wewnątrz podkwadratu.

$$\begin{aligned} (baz\_x, baz\_y) &= (x \ div \ 3^{k-1}, y \ div \ 3^{k-1}) \\ (wew\_x, wew\_y) &= (x \ mod \ 3^{k-1}, y \ mod \ 3^{k-1}) \end{aligned}$$

Rys. 1 Transformacje współrzędnych (s oznacza długość boku podkwadratu): $T1 : (x,y) \mapsto (x,y)$, $T2 : (x,y) \mapsto (s-x-1, s-y-1)$, $T3 : (x,y) \mapsto (y,x)$, $T4 : (x,y) \mapsto (s-y-1, s-x-1)$.



Wynika z tego następujący ogólny schemat liczenia odległości $DIST(k,x,y)$.

```
function DIST (k, x, y) : integer;
begin
  if k = 0 then DIST := 0
  else begin
    (baz_x, baz_y) := (x div 3^(k-1), y div 3^(k-1));
    (wew_x, wew_y) := (x mod 3^(k-1), y mod 3^(k-1));
    oblicz (x', y') jako wynik transformacji TR(baz_x, baz_y)
    zastosowanej do punktu (wew_x, wew_y);
    DIST := DIST(k-1, x', y') + NUM(baz_x, baz_y) · 3^(k-1)
  end
end
```

Zobaczmy, w jaki sposób wykonywane jest obliczanie $DIST(3,x,y)$ dla punktu $(x,y) = (9,10)$.

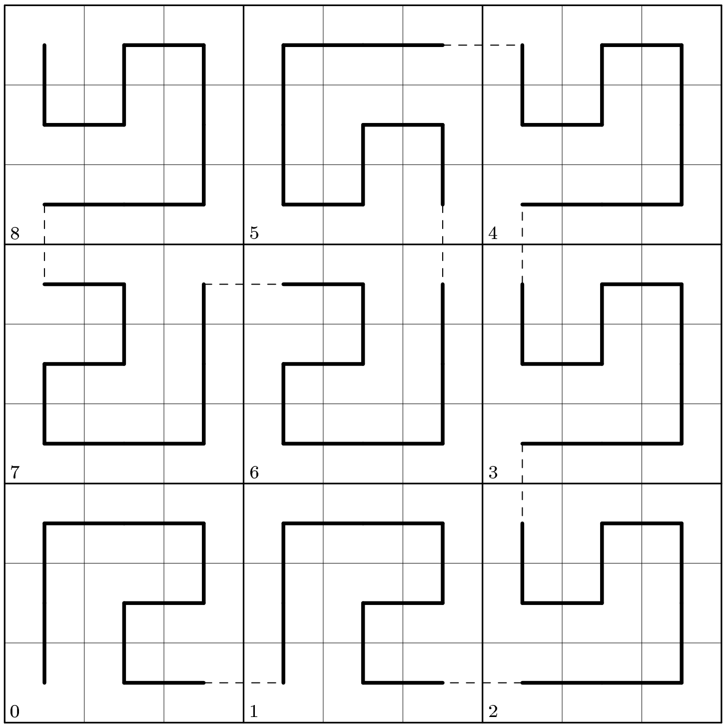
- Obliczamy $(baz\_x,\ baz\_y) := (1,1)$, $(wew\_x,\ wew\_y) := (0,1)$.
- Stosujemy transformację $TR(1,1)$ do $(0,1)$ w podkwadracie o boku $s = 9$ i otrzymujemy punkt $(x',y') = (8-1, 8-0) = (7,8)$. Mamy $NUM(1,1) = 6$. Zatem:

$$DIST(3,9,10) = DIST(2,7,8) + 6 \cdot 9^2$$

- Podobnie obliczamy $DIST(2,7,8) = DIST(1,1,2) + 4 \cdot 9$
- Ostatecznie $DIST(1,1,2) = 5$ i jako wynik otrzymujemy

$$DIST(3,9,10) = 6 \cdot 9^2 + 4 \cdot 9 + 5.$$

Rys. 2 Kolejność odwiedzania podkwadratów bazowych oraz tablice TR i NUM.



Tablica $TR[i,j]$

j			
2	T1	T2	T1
1	T4	T4	T1
0	T3	T3	T1
i	0	1	2

Tablica $NUM[i,j]$

j			
2	8	5	4
1	7	6	3
0	0	1	2
i	0	1	2

Zamiast za każdym razem pracowicie liczyć operacje **div** i **mod** wygodniej jest raz przedstawić x i y w systemie trójkowym; wtedy w każdej iteracji błyskawicznie odczytujemy pozycję podkwadratu bazowego. Tego typu chwyt algorytmiczny zastosowano w programie wzorcowym. Ma to jednak wyłącznie charakter techniczny i nie zmienia istotnie sposobu obliczania $DIST$.

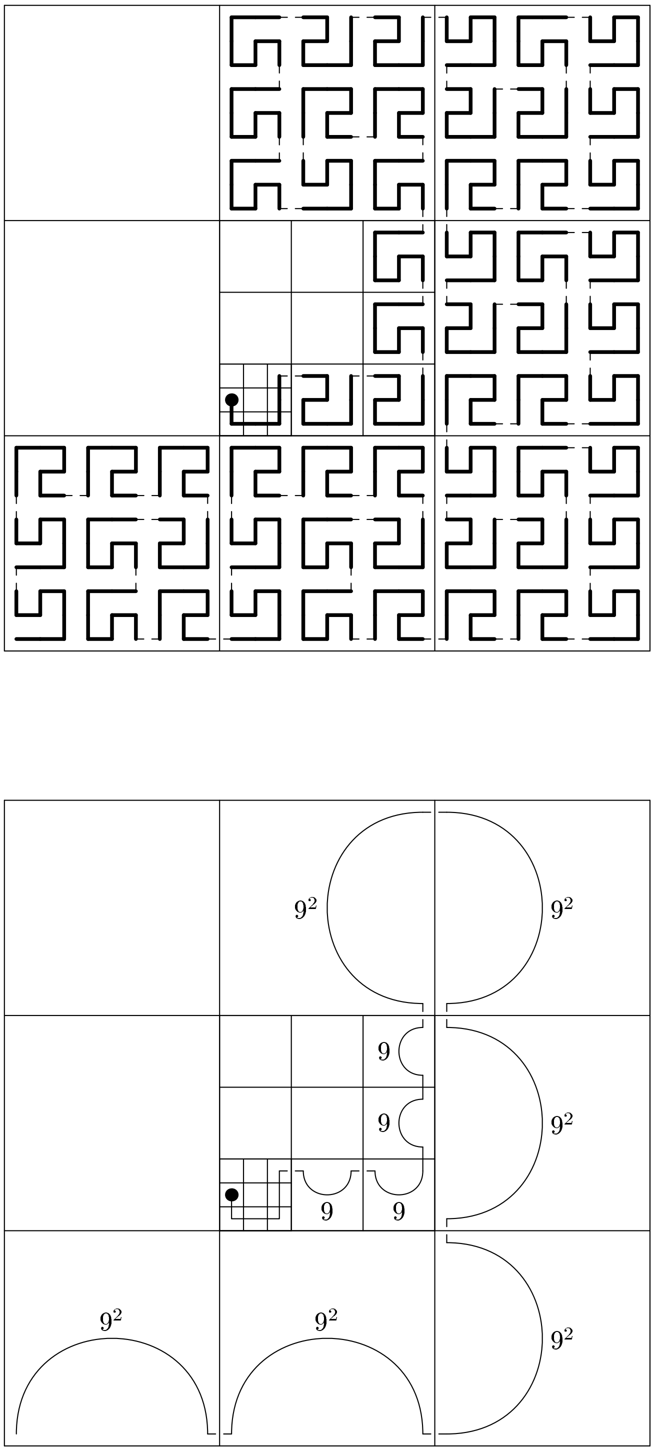
TESTY

Do oceny rozwiązań zawodników użyto 11-tu testów LUN0.IN–LUN10.IN:

- LUN0.IN — $k = 2$, test z treści zadania,
- LUN1.IN — $k = 1$, najmiejszy możliwy rozmiar dachu, lewy dolny i prawy dolny róg planszy to, odpowiednio, początek i koniec obserwowanej drogi lunatyka,

- LUN2.IN, LUN3.IN — testy poprawnościowe,
- LUN4.IN — $k = 30$, test losowy,
- LUN5.IN — $k = 60$, maksymalny rozmiar planszy, lewy dolny i prawy górny róg,
- LUN6.IN — $k = 60$, największy możliwy wynik, lewy dolny i lewy górny róg,
- LUN7.IN–LUN9.IN — testy losowe, duże k,
- LUN10.IN — $k = 6$, dwa sąsiednie pola.

Rys. 3 Symulacja drogi lunatyka do punktu $(x, y) = (9, 10)$ za pomocą „przeskakiwania” przez podkwadraty, które lunatyk przechodzi w całości (przypominamy, że współrzędne liczymy od 0). W tym wypadku „przeskakujemy” 6 podkwadratów stopnia 2 i 4 stopnia 1.



Wojciech Rytter
autor zadania, opracowanie

Adam Borowski
program wzorcowy

Rakiety

Na płaskiej mapie wyznaczono dwa rozłączne, n-elementowe zbiory punktów: R i W. Żadne trzy punkty ze zbioru $R \cup W$ nie są współliniowe. W punktach ze zbioru R rozlokowane są rakiety typu ziemia-ziemia, natomiast w punktach ze zbioru W znajdują się obiekty wroga, które trzeba zniszczyć. Rakiety mogą lecieć jedynie w linii prostej, a tory rakiet nie mogą się przecinać. Chcemy dla każdej rakiety wyznaczyć cel, który powinna zniszczyć.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `RAK.IN` *współrzędne punktów ze zbiorów R i W,*
- *wyznacza zbiór n parami nie przecinających się odcinków i takich, że jeden koniec każdego odcinka należy do zbioru R, podczas gdy drugi należy do zbioru W,*
- *zapisuje wynik w pliku tekstowym* `RAK.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `RAK.IN` *jest zapisana jedna liczba naturalna n, $1 \leqslant n \leqslant 10000$, równa liczności zbiorów R i W.*

W każdym z kolejnych $2n$ wierszy pliku `RAK.IN` *jest zapisana para liczb całkowitych z przedziału $[-10000, 10000]$ oddzielonych pojedynczym odstępem. Są to współrzędne punktu na płaszczyźnie (najpierw współrzędna x, potem współrzędna y). Pierwsze n z tych wierszy zawiera współrzędne punktów ze zbioru R, natomiast ostatnie n wierszy zawiera współrzędne punktów ze zbioru W. W wierszu o numerze $i+1$ znajdują się współrzędne punktu r_i, natomiast w wierszu o numerze $i+n+1$ znajdują się współrzędne punktu w_i, $1 \leqslant i \leqslant n$.*

Wyjście

Plik `RAK.OUT` *powinien składać się z n wierszy. W i-tym wierszu należy zapisać jedną liczbę naturalną k_i taką, że odcinek $\overline{r_i w_{k_i}}$ należy do wyznaczonego zbioru odcinków (innymi słowy, że rakieta z punktu r_i ma zniszczyć obiekt położony w punkcie w_{k_i}).*

Przykład

Dla pliku wejściowego `RAK.IN`:

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

poprawną odpowiedzią jest na przykład plik `RAK.OUT`:

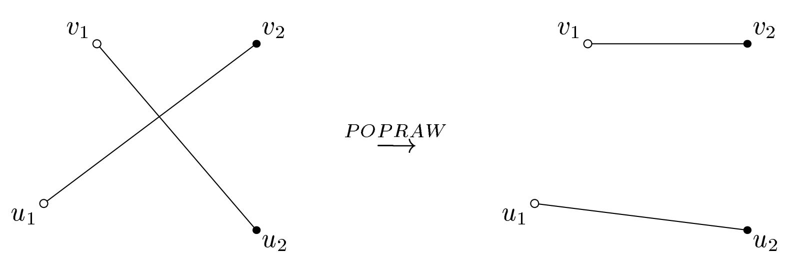
```
2
1
4
3
```

ROZWIĄZANIE

Pokolorujmy punkty z R na czarno, a punkty z W na biało. Przez proste i odcinki rozumiemy dalej tylko proste i odcinki zawierające dwa punkty różnych kolorów. Przez punkty rozumiemy tylko punkty z $R \cup W$. Naszym celem jest znalezienie n nieprzecinających się odcinków.

Jeżeli mamy dwa przecinające się odcinki $\overline{u_1 v_1}$, $\overline{u_2 v_2}$, to możemy wykonać operację *POPRAW* pokazaną na rysunku 1.

Rys. 1 Operacja $POPRAW(u_1, u_2, v_1, v_2)$.



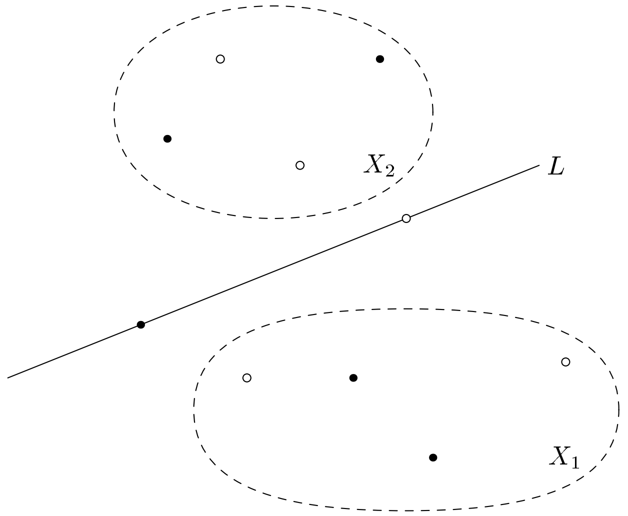
Oto **algorytm brutalny**:

```
utwórz w dowolny sposób n odcinków zawierających wszystkie punkty
while (istnieją dwa przecinające się odcinki u1v1, u2v2) do
   POPRAW(u1, u2, v1, v2)
```

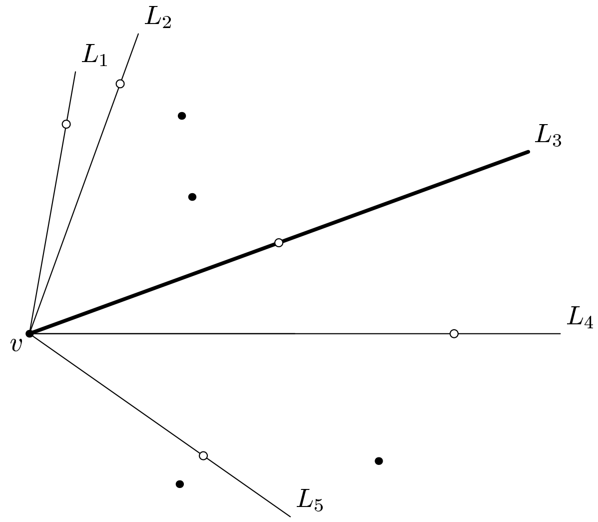
Oczywistym jest, że jeśli algorytm ten ma własność stopu, to jest poprawny. Algorytm ma własność stopu, gdyż każda wykonana operacja *POPRAW* zmniejsza sumaryczną długość odcinków. Dla wielu zestawów danych algorytm ten zadziała błyskawicznie, ale dla pewnych złośliwych przypadków może pracować bardzo długo.

Zrównoważonym zbiorem punktów będziemy nazywać taki zbiór, w którym jest tyle samo punktów czarnych, co białych. Podamy teraz algorytm korzystający z pojęcia *prostej równoważącej*. Jest to taka prosta, która dzieli zbiór niezawartych w niej punktów na dwa zbiory zrównoważone (zob. rysunek 2). Załóżmy, że mamy funkcję $PROSTA\_ROW(X)$ znajdującą prostą równoważącą dla niepustego podzbioru X. Wtedy wynik obliczamy wywołując $ODCINKI\_W\_ZBIORZE(R \cup W)$.

Rys. 2 Prosta równoważąca.



Rys. 3 Ilustracja graficzna działania funkcji $PROSTA\_ROW$, $L = L_3$.



```
procedure ODCINKI_W_ZBIORZE(X);
begin
  if X ≠ ∅ then begin
    L := PROSTA_ROW(X);
    połącz dwa  punkty leżące na L;
    niech X1, X2 będą zbiorami punktów leżących, odpowiednio,
    po jednej i po drugiej stronie L;
    wykonaj rekurencyjnie ODCINKI_W_ZBIORZE(X1);
    wykonaj rekurencyjnie ODCINKI_W_ZBIORZE(X2)
  end
end
```

Prostą równoważącą można obliczyć w następujący sposób:

```
1:  function PROSTA_ROW(X) : prosta;
2:  begin
3:    wybierz punkt v leżący najbardziej na lewo;
4:    { przyjmijmy dla uproszczenia, że v jest czarny }
5:    niech L1, L2, ..., L4n będą prostymi przechodzącymi przez v
6:    posortowanymi względem ich współczynników kierunkowych;
7:    niech L ∈ {L1, L2, ... Ln} będzie pierwszą
8:    prostą równoważącą;
9:    PROSTA_ROW := L
10: end
```

Na rysunku 3 przedstawiono ideę działania funkcji *PROSTA_ROW*.

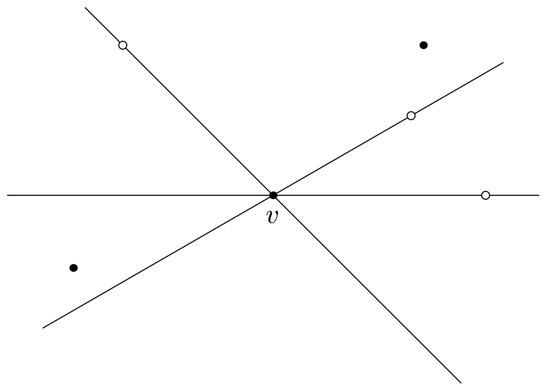
W instrukcji w wierszu 7 zawsze znajdzie się odpowiednia prosta. Wynika to z następującego faktu:

Fakt 1: Jeśli mamy ciąg punktów czarnych i białych, w którym liczba czarnych jest o jeden mniejsza niż białych, to znajdzie się punkt taki, że zarówno zbiór punktów występujących przed nim, jak i zbiór punktów występujących po nim w tym ciągu są zbiorami zrównoważonymi.

Uzasadnienie: Gdyby istniał ciąg będący kontrprzykładem, to usuwając z niego dwa sąsiednie punkty różnych kolorów otrzymalibyśmy „krótszy kontrprzykład". W końcu otrzymalibyśmy kontrprzykład długości 1. Ale ciąg długości 1, spełniający założenie, składa się tylko z punktu białego i po obu jego stronach są zbiory zrównoważone.

Obserwacja 2: Szukając prostej równoważącej zaczynamy od punktu najbardziej na lewo, natomiast nie możemy wystartować z dowolnego punktu v. Rysunek 4 pokazuje dlaczego.

Rys. 4 Nie możemy wybrać dowolnego punktu v w funkcji *PROSTA_ROW*. Nie ma równoważącej prostej przechodzącej przez punkt v na rysunku.



Funkcję *PROSTA_ROW* można łatwo zaimplementować w czasie $O(n \log n)$ (potrzeba trochę prostej geometrii: liczenie kątów). W sensie złożoności czasowej „wąskim gardłem" jest sortowanie prostych względem ich współczynników kierunkowych. Możemy pozbyć się sortowania korzystając z bardziej skomplikowanego algorytmu używającego liczenia mediany w czasie liniowym (co nie jest trywialne). Znajdujemy punkt biały w o tej własności, że po obydwu stronach prostej (v, w) jest tyle samo punktów białych (z dokładnością do 1). Jeżeli prosta (v, w) jest równoważąca, to kończymy. W przeciwnym razie poszukiwanie prostej wynikowej ograniczamy do zbioru leżącego po tej stronie prostej (v, w), po której jest mniej czarnych punktów. Dla tego zbioru powtarzamy postępowanie rekurencyjnie.

Cały algorytm ma złożoność czasową $O(n^2 \log n)$, gdy korzystamy z sortowania, oraz $O(n^2)$ gdy zastępujemy sortowanie skomplikowanym liczeniem mediany w czasie liniowym.

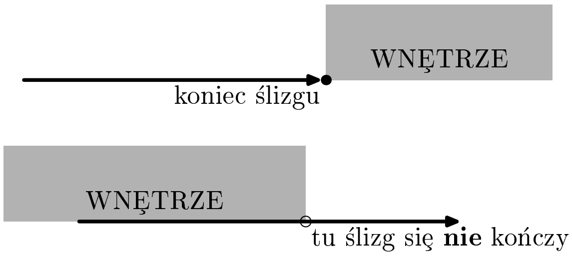
TESTY

Do oceny rozwiązań zawodników użyto 11-tu testów RAK0.IN–RAK10.IN. Największą trudnością było wygenerowanie testów dużych rozmiarów, w których żadne trzy punkty nie byłyby współliniowe. Zachęcamy czytelników do napisania efektywnego generatora takich testów.

Marcin Kubica **Adam Borowski**
autor zadania, opracowanie program wzorcowy

Lodowisko

Na największym lodowisku w Bajtocji zorganizowano zawody w ślizganiu. Lodowisko ma kształt kwadratu o rozmiarach 10000×10000. Zawodnik rozpoczyna ślizgi z punktu startowego wskazanego przez sędziów i ma za zadanie zakończyć ślizgi w punkcie docelowym, również wskazanym przez sędziów. Punkty startowy i docelowy są różne. Ślizgi mogą odbywać się tylko w kierunkach równoległych do boków lodowiska. Na lodowisku rozstawiono przeszkody. Każda przeszkoda jest graniastosłupem, którego podstawy są wielokątami o bokach równoległych do boków lodowiska. Każde dwa kolejne boki podstawy są zawsze do siebie prostopadłe. Przeszkody nie mają wspólnych punktów. Każdy ślizg kończy się przy pierwszym zetknięciu ze ścianą pewnej przeszkody, która to ściana jest prostopadła do kierunku ślizgu i ogranicza wnętrze przeszkody w kierunku tego ślizgu. Innymi słowy, zatrzymanie następuje wyłącznie na ścianie, na którą się „wpada" lub w punkcie docelowym. Wypadnięcie z lodowiska oznacza dyskwalifikację. Ślizgi mogą odbywać się wzdłuż ściany przeszkody.



Czy zawodnik wykonujący ślizgi zgodnie z podanymi regułami może dotrzeć z punktu startowego do punktu docelowego? Jesli tak, to jaka jest najmniejsza liczba ślizgów, które musi wykonać?

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `LOD.IN` *opisy lodowiska i przeszkód oraz współrzędne punktów startowego i docelowego,*
- *sprawdza, czy ślizgając się można z punktu startowego dotrzeć do punktu docelowego, a jeżeli tak, to oblicza najmniejszą liczbę ślizgów potrzebną do tego,*
- *zapisuje wynik w pliku tekstowym* `LOD.OUT`*.*

Wejście

Plan lodowiska jest naniesiony na siatkę prostokątną o rozmiarach 10000×10000. Dolny lewy róg siatki ma współrzędne $(0, 0)$. Górny prawy róg siatki ma współrzędne $(10000, 10000)$. W pierwszym wierszu pliku wejściowego `LOD.IN` *znajdują się dwie liczby całkowite z_1 i z_2 oddzielone pojedynczym odstępem, $0 \leqslant z_1, z_2 \leqslant 10000$. Para (z_1, z_2) to współrzędne punktu startowego. W drugim wierszu pliku* `LOD.IN` *znajdują się dwie liczby całkowite t_1 i t_2 oddzielone pojedynczym odstępem, $0 \leqslant t_1, t_2 \leqslant 10000$. Para (t_1, t_2) to współrzędne punktu docelowego. Trzeci wiersz pliku* `LOD.IN` *zawiera jedną liczbę naturalną s, $1 \leqslant s \leqslant 2500$. Jest to liczba przeszkód. Kolejne wiersze zawierają opisy s przeszkód. Każdy opis przeszkody rozpoczyna się od wiersza zawierającego jedną dodatnią liczbę całkowitą r równą liczbie ścian bocznych (boków podstawy) przeszkody. W kolejnych r wierszach znajdują się po dwie liczby całkowite x i y oddzielone pojedynczym odstępem. Są to kolejne współrzędne wierzchołków podstawy przeszkody przy obchodzeniu jej zgodnie z ruchem wskazówek zegara (tzn. przy obchodzeniu przeszkody wnętrze znajduje się zawsze po prawej stronie obchodzącego). Łączna liczba ścian bocznych w przeszkodach nie przekracza 10000.*

Wyjście

Twój program powinien zapisać w jednym wierszu pliku wyjściowego `LOD.OUT`*:*

- *jedno słowo 'NIE', gdy nie jest możliwe dotarcie z punktu startowego do punktu docelowego lub*

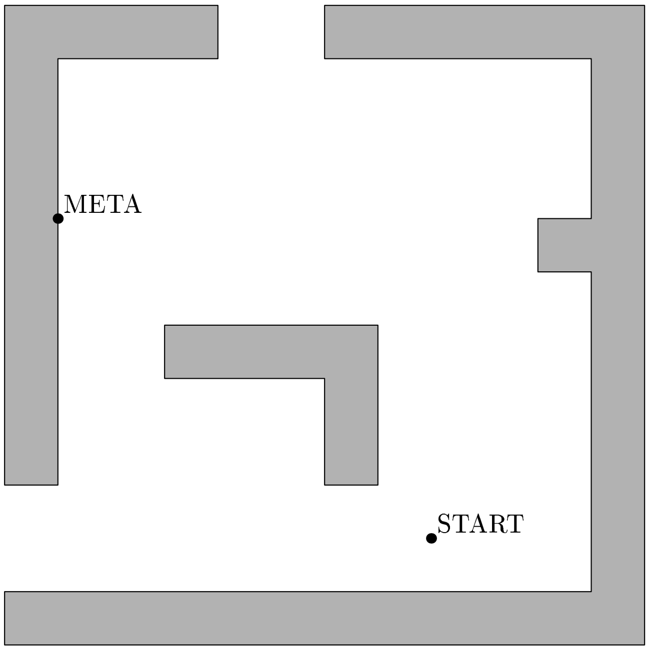
- *najmniejszą liczbę ślizgów potrzebną w tym celu, w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `LOD.IN`*:*

```
40 10
5 40
3
6
0 15
0 60
20 60
20 55
5 55
5 15
12
30 55
30 60
60 60
60 0
0 0
0 5
55 5
55 35
50 35
50 40
55 40
55 55
6
30 25
15 25
15 30
35 30
35 15
30 15
```

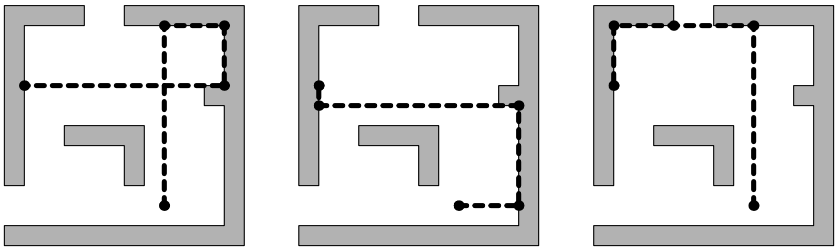
opisującego następujący układ przeszkód:



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

```
4
```

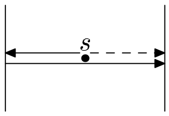
Oto możliwe ciągi ślizgów długości 4:



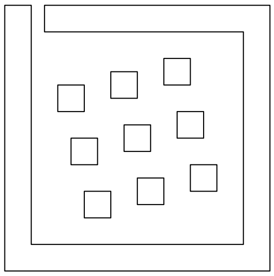
ROZWIĄZANIE

Stosunkowo łatwo można zauważyć, że mamy tu do czynienia z pewnym grafem. Wierzchołkami tego grafu są możliwe punkty zatrzymania się zawodnika oraz punkt startowy i docelowy, a krawędzie odpowiadają możliwym ślizgom. Nasze zadanie polega na znalezieniu (długości) najkrótszej ścieżki łączącej punkt startowy z docelowym. Możemy posłużyć się w tym celu algorytmem przeszukiwania grafu wszerz (por. [18], p. 2.3, [9] p. 1.5.3, [12] p. 23.2). Trudność tego zadania polega jednak na konstrukcji grafu.

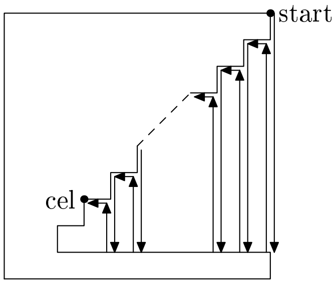
Zastanówmy się, ile wierzchołków może mieć nasz graf. Niech n będzie liczbą wierzchołków przeszkód ($n \leqslant 10\,000$). Ile może być możliwych punktów zatrzymania nie będących wierzchołkami przeszkód? Zapomnijmy na chwilę o punkcie startowym i docelowym. Zauważmy, że jeżeli zatrzymaliśmy się nie w wierzchołku przeszkody, to bez straty ogólności możemy założyć, że zaczęliśmy ten ślizg wzdłuż pewnej ściany. Jeżeli tak nie jest, to znaczy, że poprzedni ślizg został wykonany w przeciwną stronę, po czym „odbiliśmy się” od ściany. Jest to możliwe jedynie wtedy, gdy poprzedni ślizg zaczynał się w punkcie startowym. Wówczas jednak mogliśmy od razu zacząć ślizgać się w drugą stronę.



Powtarzając to rozumowanie odpowiednią liczbę razy znajdziemy taką ścianę, że ślizgając się wzdłuż niej zatrzymamy się w danym punkcie. Zauważmy, że stojąc przy ścianie możemy zacząć się ślizgać równolegle do niej tylko w dwóch kierunkach. Jeśli więc każdej ścianie przyporządkujemy punkty (leżące poza nią) w których zatrzymamy się, jeżeli zaczniemy się ślizgać równolegle do niej, to każdej ścianie przyporządkujemy co najwyżej dwa punkty. Jest to jednak oszacowanie górne, którego nie da się osiągnąć. Zaczynając ślizg przy skrajnie lewej ścianie i ślizgając się równolegle do niej wypadniemy poza lodowisko (jeśli jest kilka skrajnie lewych ścian, to wypadniemy poza lodowisko zaczynając ślizg przy najwyższej z nich i ślizgając się w górę oraz przy najniższej z nich i ślizgając się w dół). Podobnie dla skrajnie prawej, skrajnie górnej i skrajnie dolnej ściany przeszkód. Stąd punktów zatrzymania nie będących wierzchołkami przeszkód może być co najwyżej $2n-8$. Uwzględniając punkt startowy i docelowy musimy doliczyć jeszcze co najwyżej sześć punktów — punkt startowy i docelowy też mogą być wierzchołkami naszego grafu, a dodatkowo z punktu startowego możemy zacząć się ślizgać w dowolnym z czterech kierunków. Tak więc punktów zatrzymania może być co najwyżej $3n-2$. Poniższy rysunek pokazuje, jak można się zbliżyć do tego oszacowania.

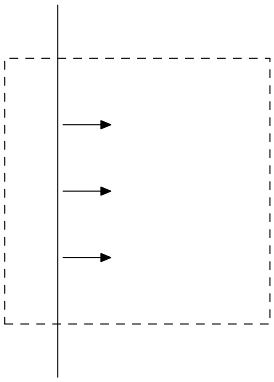


Jednym z błędów, jaki można było popełnić, było zbyt małe ograniczenie na liczbę punktów zatrzymania, np. 10 000 lub 20 000. Również liczba ślizgów niezbędnych, aby dotrzeć do punktu docelowego, może być większa od 10 000. Ilustruje to poniższy rysunek.



Najprostsze rozwiązanie, jakie się nasuwa, polega na skonstruowaniu grafu, którego wierzchołkami są możliwe punkty zatrzymania, a krawędzie odpowiadają możliwym ślizgom, oraz wyznaczeniu minimalnej liczby ślizgów potrzebnych do osiągnięcia punktu docelowego za pomocą przeszukiwania wszerz. Zarówno liczba wierzchołków, jak i krawędzi grafu jest liniowa ze względu na liczbę wierzchołków przeszkód n. Możemy więc przeszukać graf w czasie $O(n)$. Jednak konstrukcja grafu jest bardziej kosztowna. Kluczowym jej elementem, zarówno przy wyznaczaniu wierzchołków jak i krawędzi grafu, jest obliczenie punktów zatrzymania dla ślizgu o zadanym kierunku i punkcie początkowym. Najprościej możemy tego dokonać przeglądając za każdym razem wszystkie ściany prostopadłe do kierunku ślizgu. Wówczas jednak nasz program będzie działał w czasie rzędu $O(n^2)$.

Algorytm ten możemy przyspieszyć korzystając z techniki zamiatania (por. [9] p. 8.4, [12] p. 35.2). Opiera się ona na następującym pomyśle. Mamy dokonać pewnych obliczeń geometrycznych na płaszczyźnie. Po płaszczyźnie „przesuwamy” prostą (zwaną miotłą) — przykładowo niech to będzie prosta pionowa, którą przesuwamy z lewa na prawo. Staramy się zredukować nasz problem rozwiązując go w danej chwili tylko w obrębie miotły. Ponadto nie rozważamy każdego możliwego położenia miotły, tylko przesuwając ją „zatrzymujemy” się, gdy miotła napotyka punkty istotne dla obliczeń. Zwykle z miotłą związana jest pewna struktura danych, którą aktualizujemy w miarę przesuwania miotły.



Za pomocą jednego zamiatania możemy np. wyznaczyć dla każdej pionowej ściany punkty zatrzymania dla pionowych ślizgów zaczynających się przy tych ścianach. W strukturze danych odpowiadającej miotle przechowujemy poziome ściany przecinające miotłę (a dokładniej ich współrzędne Y). Miotłę zatrzymujemy, gdy napotyka ona wierzchołki ścian. Początkowo miotła jest pusta. Gdy miotła napotyka pewne wierzchołki przeszkód — są to równocześnie końce pewnej pionowej ściany (ścian), jak i końce poziomych ścian — wykonujemy dla nich następujące operacje (w podanej kolejności):

- dla każdego wierzchołka będącego lewym końcem poziomej ściany wstawiamy tę ścianę do miotły,
- dla każdego wierzchołka leżącego we wklęsłym kącie ściany (kąt) ślizg zaczynający się przy pionowej ścianie przylegającej do tego wierzchołka i skierowany przez ten wierzchołek zatrzyma się w nim,
- dla każdego wierzchołka leżącego w wypukłym kącie ściany (narożnik) ślizg zaczynający się przy pionowej ścianie przylegającej do tego wierzchołka i skierowany przez ten wierzchołek zatrzyma się na najbliższej poziomej ścianie należącej do miotły, leżącej, odpowiednio, powyżej lub poniżej danego wierzchołka,
- dla każdego wierzchołka będącego prawym końcem poziomej ściany usuwamy tę ścianę z miotły.

Aby wyznaczyć miejsca zatrzymania miotły, przed przystąpieniem do zamiatania sortujemy wierzchołki przeszkód względem ich współrzędnych X. Współrzędne te są liczbami całkowitymi od 0 do 10 000 — możemy więc zastosować np. sortowanie przez zliczanie o koszcie $O(n)$.

Stosując drugie zamiatanie (w pionie) wyznaczamy pozostałe wierzchołki naszego grafu. Nie mamy jednak wyznaczonych wszystkich jego krawędzi. Lukę tę uzupełnimy wprowadzając dwie poprawki do opisanego algorytmu.

Po pierwsze, jeżeli zaczynamy ślizg równoległy do ściany, przy której stoimy i zatrzymujemy się w punkcie, który leży na innej ścianie, lecz nie na jej końcu, to odbijając się od niej prostopadle zatrzymamy się w wierzchołku ściany, przy której staliśmy. Podobnie, jeżeli zaczynając ślizg z punktu startowego zatrzymamy się na pewnej ścianie, to odbijając się od niej prostopadle zatrzymamy się w tym samym punkcie, co gdybyśmy z punktu startowego ślizgali się w przeciwnym kierunku.

Po drugie, w przypadku ślizgów równoległych do ściany, przy której stoimy, punkt zatrzymania nie zależy od tego, w którym miejscu ściany zaczynaliśmy ślizg. Możemy więc dla każdego punktu zatrzymania nie będącego wierzchołkiem ściany pamiętać, na jakiej ścianie on leży, po czym poprowadzić z tego punktu krawędzie do punktów zatrzymania, dla ślizgów równoległych do ściany zaczynających się w jej końcach.

Efektywność opisanego algorytmu zależy od sposobu zaimplementowania miotły i operacji na niej. Miotła jest tu zbiorem liczb całkowitych (z przedziału od 0 do 10 000) z następującymi operacjami:

- inicjuj miotłę jako pusty zbiór,
- wstaw liczbę do zbioru,
- usuń liczbę ze zbioru,
- dla zadanej liczby znajdź najmniejszą liczbę większą od niej i należącą do miotły,

- dla zadanej liczby znajdź największą liczbę mniejszą od niej i należącą do miotły.

Jeżeli zaimplementujemy miotłę jako listę, to koszt czasowy operacji na niej będzie wynosił $O(n)$, a cały algorytm będzie działał w czasie $O(n^2)$. Mimo, że jest to ten sam rząd co w przypadku poprzedniego algorytmu, to takie rozwiązanie ma szansę być bardziej efektywne, gdyż miotła zwykle zawiera istotnie mniej niż $(n/2)$ ścian.

Możemy tu jednak zastosować jedną z wielu słownikowych struktur danych realizującą powyższe operacje w czasie $O(\log n)$. Daje to koszt całkowity algorytmu rzędu $O(n \log n)$.

Pozwólmy sobie na drobną dygresję. W przypadku prawie wszystkich zadań olimpijskich podawane jest górne ograniczenie na rozmiar danych wejściowych. W przypadku wielu zadań należy wykorzystać taką czy inną strukturę służącą do przechowywania danych, przy czym rozmiar tej struktury potrafimy ograniczyć na podstawie ograniczenia rozmiaru danych wejściowych. W takich przypadkach należy się zastanowić nad wykorzystaniem struktury danych będącej drzewem zrównoważonym, często o ustalonym kształcie, pamiętanym w statycznej tablicy. Podstawowe operacje na takich strukturach są zwykle proste do zaimplementowania, a działają w czasie $O(\log n)$.

Najbardziej znanym przykładem takiej struktury danych jest kopiec (por. [9] p. 2.6, [12] r. 7). Pozwala on na wstawianie elementów i usuwanie elementu najmniejszego (największego) w czasie $O(\log n)$. Kopiec jest w pełni zrównoważonym drzewem binarnym, w którego węzłach są przechowywane liczby, przy czym wartość przechowywana w ojcu jest mniejsza (większa) od wartości przechowywanych w jego synach. Istotny jest sposób pamiętania kopca w tablicy. Kształt kopca n-elementowego jest ustalony i zajmuje on pierwszych n elementów tablicy (indeksowanej od 1). Wartość przechowywana w korzeniu znajduje się pod indeksem 1. Niech wartość przechowywana w danym węźle znajduje się w i-tym elemencie tablicy. Jeżeli $2i > n$, to dany węzeł jest liściem. Jeżeli $2i = n$, to dany węzeł ma jednego syna, którego wartość jest pamiętana w n-tym elemencie tablicy. Jeżeli $2i < n$, to dany węzeł ma dwóch synów, a ich wartości są pamiętane w $2i$-tym i $(2i+1)$-szym elemencie tablicy. Jak łatwo zauważyć, wartość przechowywana w ojcu danego węzła znajduje się w $\lfloor (i/2) \rfloor$-tym elemencie tablicy.

Innym przykładem takiej struktury jest drzewo BST o ograniczonej wysokości. Przechowujemy je w tablicy, podobnie jak kopiec, przy czym elementy tablicy zawierają, oprócz wartości przechowywanych w węzłach, znaczniki wskazujące, czy dany element jest wykorzystany. Strukturę tę stosujemy, gdy najpierw znamy wszystkie elementy, które należy wstawić do drzewa, po czym są one sukcesywnie z niego usuwane. Po posortowaniu elementów tworzymy w pełni zrównoważone drzewo BST. Usuwanie elementów nie zwiększa wysokości drzewa, więc możemy to robić tak samo, jak w przypadku zwykłych drzew BST, zaznaczając zwalniane elementy.

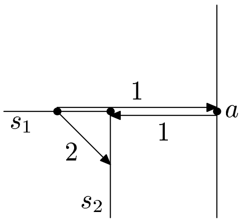
W naszym przypadku możemy skorzystać z drzewa całkowitoliczbowego. Jest to struktura służąca do pamiętania zbiorów (lub multizbiorów) złożonych z liczb całkowitych z określonego przedziału (powiedzmy k-elementowego). Jest to drzewo binarne o głębokości $\lceil \log_2 k \rceil$, mające wszystkie możliwe węzły do głębokości $\lceil \log_2 k \rceil - 1$ oraz k liści na głębokości $\lceil \log_2 k \rceil$. W węzłach tego drzewa przechowujemy liczby całkowite (od 0 do k). Liście znajdujące się najgłębiej reprezentują elementy należące do zbioru — jeżeli element należy do zbioru, to w liściu znajduje się 1, a w przeciwnym przypadku 0. W każdym węźle wewnętrznym pamiętamy sumę wartości przechowywanych w liściach poddrzewa o korzeniu w tym węźle. Inaczej mówiąc, w każdym węźle wewnętrznym pamiętamy sumę wartości przechowywanych w jego dzieciach. Drzewo takie pamiętamy, podobnie jak kopiec, w $2^{\lceil \log_2 k \rceil} - 1 + k$-elementowej tablicy liczb całkowitych. Drzewo całkowitoliczbowe pozwala nam na wykonywanie następujących operacji:

- inicjacja drzewa — w czasie $O(k)$,
- wstawienie elementu — w czasie $O(\log k)$,
- usunięcie elementu — w czasie $O(\log k)$,
- wybór najmniejszego (największego) elementu w zbiorze — w czasie $O(\log k)$,
- wybór najmniejszego (największego) elementu w zbiorze większego (mniejszego) od zadanej liczby — w czasie $O(\log k)$,
- podanie liczby elementów w zbiorze — w czasie $O(1)$.

Implementując miotłę jako drzewo całkowitoliczbowe i stosując zamiatanie otrzymujemy rozwiązanie działające w czasie $O(n \log n)$. Nie jest to jednak jeszcze rozwiązanie wzorcowe.

Główną wadą takiego rozwiązania jest zbyt duży rozmiar konstruowanego grafu. Postaramy się zastąpić punkty zatrzymania, jako wierzchołki grafu, ścianami na których się zatrzymujemy. Przypomnijmy, że w przypadku ślizgu równoległego do ściany, przy której stoimy nie jest istotne, z którego dokładnie punktu przy tej ścianie zaczynamy ślizg. W przypadku ślizgu prostopadłego do ściany (nie zaczynającego się w punkcie startowym) możliwe są następujące przypadki:

- zaczynamy ślizg w wierzchołku ściany — wówczas jest to ślizg równoległy do drugiej ściany o końcu w tym wierzchołku,
- punkt startowy nie leży przy żadnej ścianie, zaczynamy ślizg w punkcie, który nie jest wierzchołkiem i do którego dostaliśmy się jednym ślizgiem z punktu startowego — przypadek taki możemy pominąć, gdyż ma on o jeden ślizg więcej niż gdybyśmy z punktu startowego ślizgali się w przeciwnym kierunku,
- punkt startowy leży przy ścianie, ale nie w wierzchołku, zaczynamy ślizg w punkcie, który nie jest wierzchołkiem i do którego dostaliśmy się jednym ślizgiem z punktu startowego, prostopadłym do ściany, przy której leży punkt startowy — przypadek taki możemy pominąć, gdyż prowadzi on nas z powrotem do punktu startowego,
- zaczynamy ślizg w punkcie a, który nie jest wierzchołkiem i do którego dostaliśmy się w wyniku ślizgu zaczynającego się przy pewnej ścianie s_1, równoległego do niej — wówczas drugi ślizg kończy się w wierzchołku łączącym ściany s_1 i s_2.



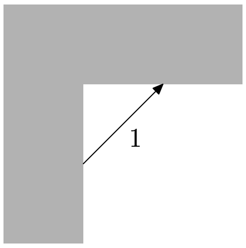
Możemy wówczas „zapomnieć” o punkcie a, pamiętając jedynie, że w dwóch ślizgach możemy się dostać z punktu leżącego przy ścianie s_1 do punktu leżącego przy ścianie s_2.

Możemy więc posłużyć się grafem, którego wierzchołkami są ściany przeszkód, a krawędzie mają długość 1 lub 2. Dodatkowo dodajemy trzy specjalne wierzchołki — dwa reprezentujące punkt startowy (dla ślizgów pionowych i poziomych) i jeden reprezentujący punkt docelowy. Krawędź długości 1 prowadząca od jednej ściany do drugiej oznacza, że stojąc przy pierwszej ścianie i ślizgając się równolegle do niej, zatrzymujemy się na drugiej ścianie. Krawędź długości 2 prowadząca od jednej ściany do drugiej oznacza, że stojąc przy pierwszej ścianie i ślizgając się równolegle do niej, a następnie odbijając się prostopadle od pewnej ściany, zatrzymujemy się na drugiej ścianie. Graf taki ma co najwyżej 10 003 wierzchołki, a z każdego wierzchołka wychodzą co najwyżej dwie krawędzie długości 1 i dwie krawędzie długości 2.

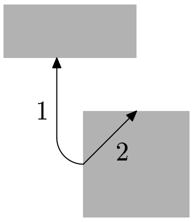
Zastanówmy się, jak można wyznaczyć krawędzie tego grafu. Podobnie jak poprzednio korzystamy dwukrotnie z zamiatania — pionowo i poziomo. Zamiatając poziomo tworzymy krawędzie odpowiadające pionowym ślizgom, a zamiatając pionowo, odpowiadające poziomym ślizgom. Dla ustalenia uwagi opiszemy poziome zamiatanie.

Struktura danych miotły zawiera ściany przecinające prostopadle miotłę, uporządkowane wedle ich współrzędnych Y. Początkowo miotła jest pusta. Strukturę tę implementujemy jako drzewo całkowitoliczbowe oraz tablicę zawierającą informację, która ściana przecina w danym miejscu miotłę. Wierzchołki ścian sortujemy (przez zliczanie — w czasie $O(n)$) ze względu na ich współrzędne X. Miotłę zatrzymujemy na wierzchołkach ścian. Gdy miotła napotyka pewne wierzchołki przeszkód, wykonujemy dla nich następujące operacje:

- dla każdego wierzchołka będącego lewym końcem poziomej ściany wstawiamy tę ścianę do miotły,
- jeżeli punkt docelowy leży na miotle, to wstawiamy do miotły odpowiadający mu wierzchołek,
- dla każdego wierzchołka leżącego we wklęsłym kącie ściany (kąt) prowadzimy krawędź długości 1, od pionowej ściany o końcu w tym wierzchołku, do poziomej ściany o końcu w tym wierzchołku,



- dla każdego wierzchołka leżącego w wypukłym kącie ściany (narożnik), w zależności od tego, czy jest to górny, czy dolny koniec pionowej ściany, znajdujemy najbliższą poziomą ścianę przecinającą miotłę (lub punkt docelowy) i leżącą, odpowiednio, powyżej lub poniżej niego; następnie prowadzimy krawędź długości 1 od pierwszej ściany do drugiej, oraz krawędź długości 2 od pierwszej ściany do ściany stykającej się z nią w danym wierzchołku,



- dla punktu startowego znajdujemy najbliższe poziome ściany przecinające miotłę (lub punkt docelowy), leżące powyżej i poniżej; następnie prowadzimy krawędź długości 1 od punktu startowego do tych ścian,
- dla każdego wierzchołka będącego prawym końcem poziomej ściany, usuwamy tę ścianę z miotły,
- jeżeli punkt docelowy leży na miotle, to usuwamy z miotły odpowiadający mu wierzchołek.

Zamiatania wykonują się w czasie $O(n \log n)$. Minimalną liczbę potrzebnych ślizgów wyznaczamy za pomocą zmodyfikowanego algorytmu przeszukiwania wszerz. (Możemy też posłużyć się algorytmem Dijkstry, por. [18] p. 3.3, [9] p. 7.6, [12] p. 25.2. Jest to jednak mniej efektywne i bardziej skomplikowane rozwiązanie.) W przypadku zwykłego przeszukiwania wszerz posługujemy się kolejką wierzchołków grafu. W naszym przypadku posługujemy się trzema kolejkami — pierwsza zawiera aktualnie przeglądane wierzchołki, do drugiej wstawiamy wierzchołki, do których prowadzą krawędzie długości 1, a do trzeciej — wierzchołki, do których prowadzą krawędzie długości 2. Na początku do pierwszej kolejki wstawiamy wierzchołki reprezentujące punkt startowy. W momencie, gdy pierwsza kolejka staje się pusta, przenosimy do niej wierzchołki z drugiej kolejki, a do drugiej kolejki przenosimy wierzchołki z trzeciej kolejki. Jednocześnie zwiększa się o 1 odległość od punktu startowego wierzchołków znajdujących się w pierwszej kolejce. Przeszukiwanie kończymy, gdy osiągniemy punkt docelowy, lub gdy wszystkie trzy kolejki będą puste. (W ostatnim przypadku punktu docelowego nie można osiągnąć.)

Przeszukiwanie grafu działa w czasie $O(n)$. Tak więc cały algorytm działa w czasie $O(n \log n)$. Rozwiązanie to znajduje się na załączonej dyskietce.

TESTY

Do oceny rozwiązań zawodników użyto 26-ciu testów. Test LOD0.IN był testem z treści zadania. Testy dla tego zadania zostały podzielone na grupy. Rozwiązanie uzyskiwało punkty za testy z danej grupy tylko wtedy, gdy przechodziło wszystkie testy z tej grupy. Dzięki temu rozwiązania podające stale wynik 'NIE' nie otrzymywały żadnych punktów. Testy należące do tej samej grupy mają w nazwie tę samą liczbę (numer grupy) z ew. dodaną literą 'a' lub 'b'.

Testy z grupy 1 i 2 to proste testy oceniające poprawność rozwiązania.

Testy z grup 3–5 i 11–12 to testy charakteryzujące się dużą liczbą możliwych punktów zatrzymania oraz stosunkowo niewielką długością ścieżki prowadzącej do punktu docelowego (o ile taka ścieżka istnieje).

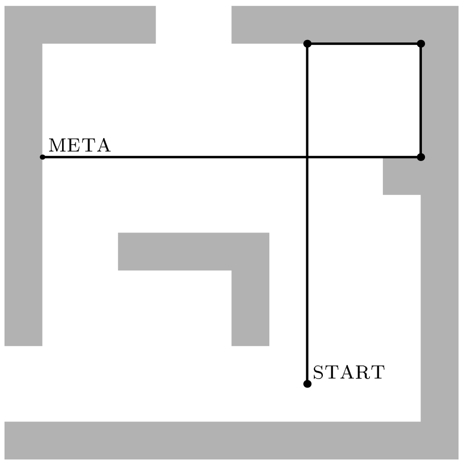
Niektóre z nich to rysunki zaadoptowane jako testy. Testy te miały na celu wychwycenie prostych, acz nieefektywnych rozwiązań.

Testy z grup 7–8, 10 i 13 to testy o zróżnicowanej liczbie możliwych punktów zatrzymania i różnej długości ścieżek prowadzących do punktów docelowych (o ile ścieżki takie istnieją), acz nie większej niż 10 000 ślizgów. Testy te miały na celu zróżnicowanie rozwiązań ze względu na ich efektywność. Testy nr 6 i 9 charakteryzują się dużą liczbą możliwych punktów zatrzymania oraz ścieżką prowadzącą do punktu docelowego dłuższą niż 10 000 ślizgów. Testy te sprawdzają również efektywność rozwiązań.

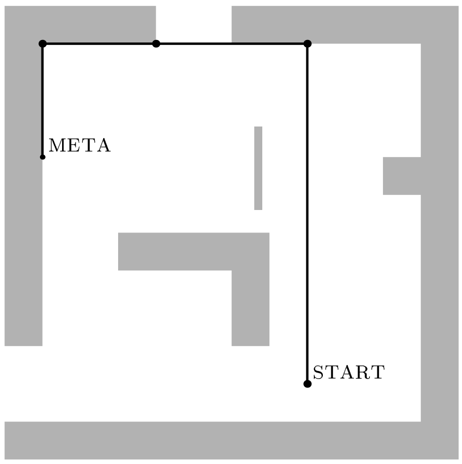
W poniższej tabeli są zestawione wielkości liczbowe charakteryzujące testy.

Test	**L. ścian**	**Wynik**
1	328	NIE
1a	328	13
2	326	32
2a	124	NIE
3	7 870	398
3a	7 870	NIE
4	9 804	195
5	9 988	65
6	9 396	14 060
7	10 000	2 500
7a	10 000	2 500
7b	10 000	2 500

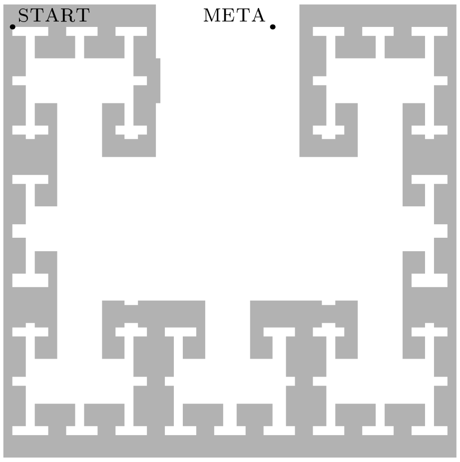
Test	**L. ścian**	**Wynik**
8	2 506	484
8a	4 290	NIE
9	10 000	14 987
9a	12	3
10	8 886	211
10a	10 000	2 623
10b	9 996	4 995
11	9 994	629
11a	10 000	NIE
12	7 730	404
12a	9 054	444
13	7 606	2 851
13a	7 606	NIE



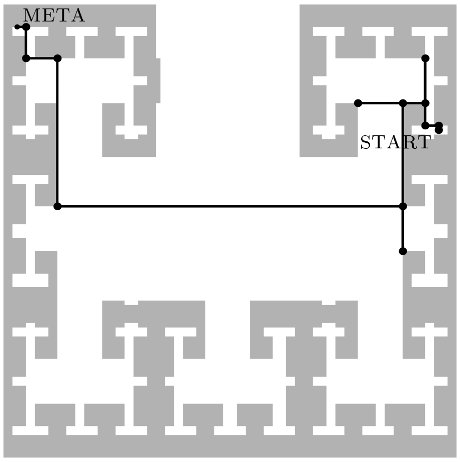
LOD0.IN



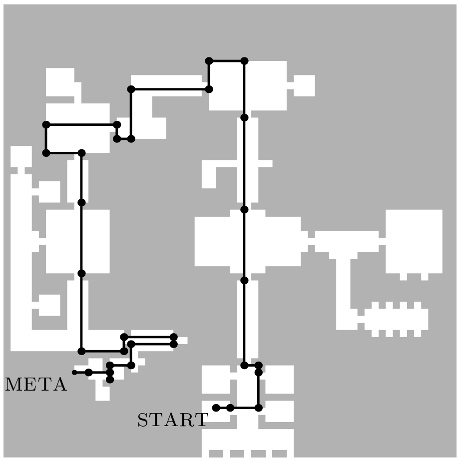
LOD0A.IN



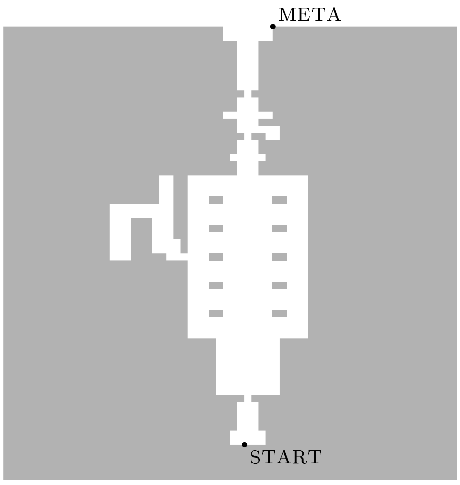
LOD1.IN



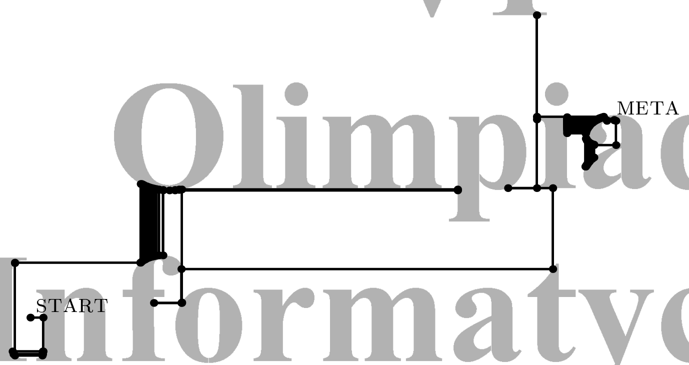
LOD1A.IN



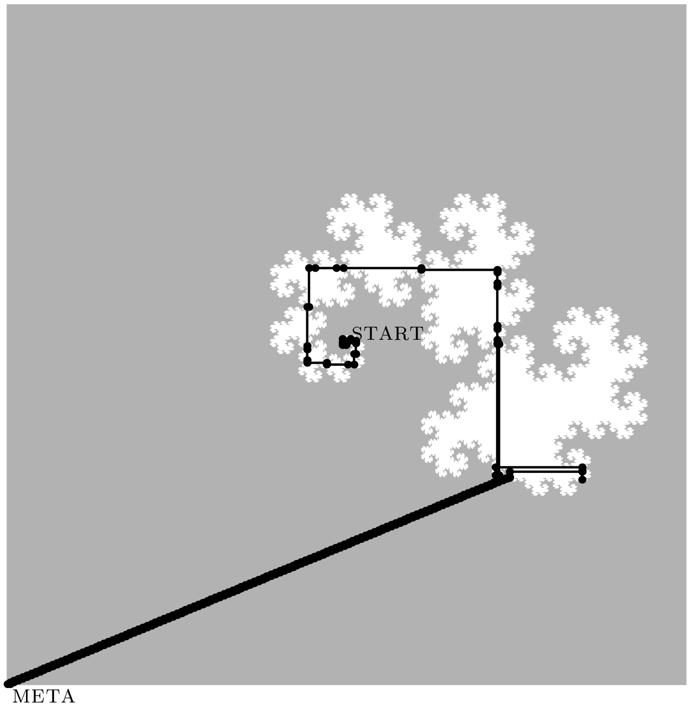
LOD2.IN



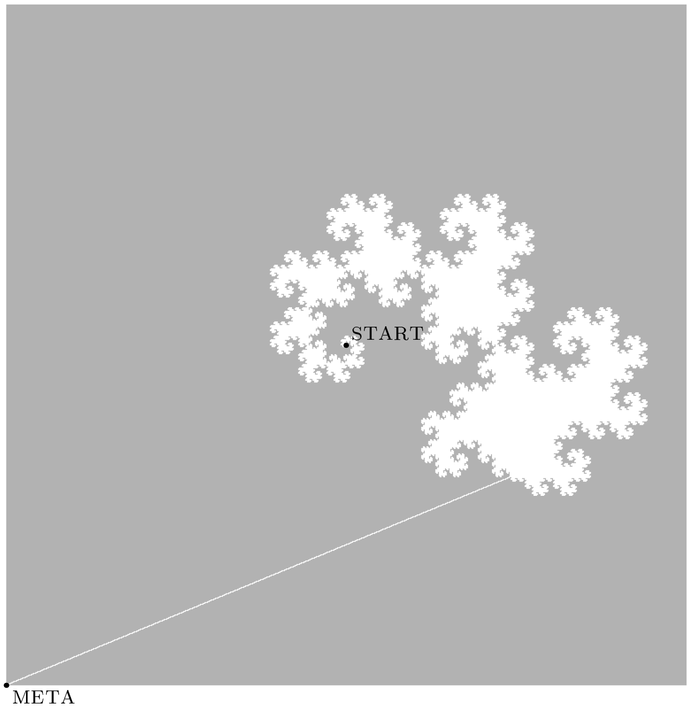
LOD2A.IN



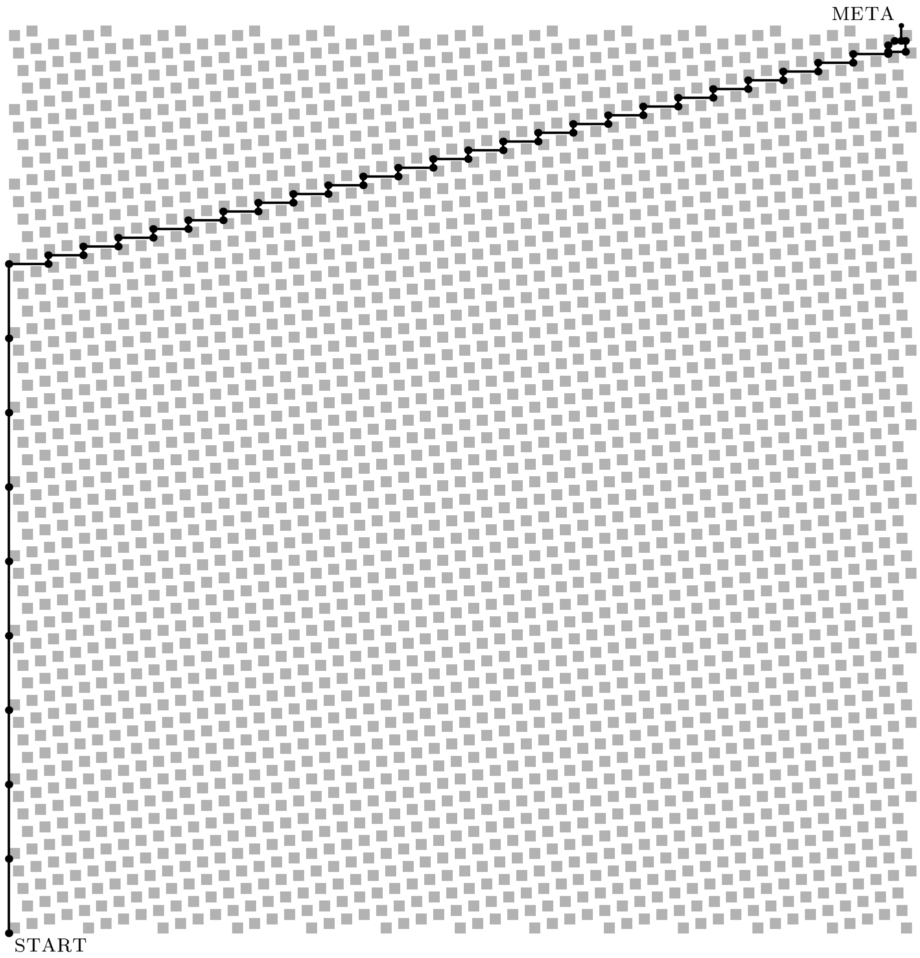
LOD4.IN



LOD3.IN



LOD3A.IN



LOD5.IN