Zawody III stopnia

opracowania zadań

Krzysztof Loryś
autor zadania, opracowanie

Marcin Stefaniak
program wzorcowy

Magazynier

Podłoga magazynu jest prostokątem podzielonym na $n \times m$ kwadratowych pól. Dwa pola są sąsiednie, *jeśli mają wspólny bok. Na jednym z pól stoi paczka. Każde inne pole jest albo wolne, albo zajęte przez skrzynię, której magazynier nie jest w stanie poruszyć. Magazynier musi przepchnąć paczkę z pola początkowego P na pole docelowe K. Magazynier może przemieszczać się po wolnych polach, przechodząc z pola, na którym stoi, na dowolne z wolnych pól sąsiednich. Jeśli magazynier stoi na polu sąsiadującym z polem, na którym jest paczka, to może ją przepchnąć na pole sąsiednie z przeciwnej strony paczki, jeżeli jest ono wolne.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `MAG.IN` *plan magazynu, początkowe położenia magazyniera i paczki oraz docelowe położenie paczki,*
- *obliczy minimalną liczbę przesunięć paczki przez granice pól, wymaganą do ustawienia jej w położeniu docelowym lub stwierdzi, że to jest niemożliwe,*
- *zapisze wynik w pliku tekstowym* `MAG.OUT`.

Wejście

W pierwszym wierszu pliku wejściowego `MAG.IN` *są zapisane dwie dodatnie liczby całkowite n i m oddzielone pojedynczym odstępem, $n, m \leqslant 100$. Są to rozmiary magazynu. W każdym z kolejnych n wierszy znajduje się jedno m-literowe słowo utworzone z liter* `S`, `M`, `P`, `K`, `w`. *Litera na i-tej pozycji j-tego z tych słów oznacza stan pola o współrzędnych (i, j):*

- `S` — *skrzynia,*
- `M` — *początkowa pozycja magazyniera,*
- `P` — *początkowa pozycja paczki,*
- `K` — *docelowa pozycja paczki,*
- `w` — *wolne pole.*

Każda z liter `M`, `P` *i* `K` *występuje w pliku* `MAG.IN` *dokładnie raz.*

Wyjście

Twój program powinien zapisać w pliku wyjściowym `MAG.OUT`*:*

- *dokładnie jedno słowo* `NIE`, *jeżeli paczki nie da się umieścić w położeniu docelowym,*
- *dokładnie jedną liczbę całkowitą, równą minimalnej liczbie przesunięć paczki przez granice pól, wymaganą do umieszczenia paczki w położeniu docelowym, jeżeli paczkę da się tam przepchnąć.*

Przykład

Dla pliku wejściowego `MAG.IN`*:*

```
10 12
SSSSSSSSSSSS
SwwwwwwwSSSS
SwSSSSwwSSSS
SwSSSSwwSKSS
SwSSSSwwSwSS
```

```
SwwwwwPwwwww
SSSSSSSwSwSw
SSSSSSMwSwww
SSSSSSSSSSSS
SSSSSSSSSSSS
```

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

```
7
```

ROZWIĄZANIE

Zadanie oparte jest na popularnej swego czasu grze komputerowej *Sokoban*. Scenariusz gry jest taki sam, jak w naszym zadaniu: mamy magazyn, magazyniera i paczki; ten sam jest cel: paczki należy przesunąć na wyznaczone pola docelowe; identyczne są zasady ruchów: paczki można tylko pchać i to tylko w sytuacji, gdy za paczką znajduje się wolna przestrzeń. Jedyna różnica tkwi w liczbie paczek: w Sokobanie liczba ta nie jest z góry niczym ograniczona (oczywiście poza powierzchnią magazynu), w naszym zadaniu została ograniczona do 1. Ograniczenie to ma kolosalny wpływ na złożoność problemu. Udowodniono, że w wersji bez ograniczenia liczby paczek, stwierdzenie, czy wszystkie paczki można przesunąć na wyznaczone pozycje, jest problemem NP-zupełnym, co oznacza, że nie należy spodziewać się istnienia algorytmu rozwiązującego ten problem w czasie wielomianowym (względem rozmiaru powierzchni magazynu). Istnieje natomiast prosty algorytm, który działa w czasie wielomianowym, gdy liczba paczek k jest ograniczona przez stałą. Opiszemy pokrótce ten algorytm dla $k = 1$, a następnie przedstawimy modyfikacje prowadzące do liniowego czasu działania.

Stanem gry nazwiemy parę pól magazynu takich, że na pierwszym polu znajduje się magazynier, a na drugim — paczka. Stan nazwiemy *końcowym*, jeśli pole z paczką jest polem docelowym. Możliwy przebieg gry wygodnie jest opisać w terminologii grafowej. Konstruujemy graf skierowany G_S, którego wierzchołkami są stany gry. Z wierzchołka u prowadzimy krawędź do wierzchołka v wtedy i tylko wtedy, gdy z u można przejść do v w jednym ruchu magazyniera. Pytanie, czy można paczkę przesunąć na pozycję docelową, sprowadza się do pytania o istnienie ścieżki prowadzącej od stanu początkowego do stanu końcowego. Gdy zainteresowani jesteśmy obliczeniem najmniejszej liczby przesunięć paczek, możemy nadać krawędziom wagi: 1 — jeśli odpowiada ona przejściu między stanami z przesunięciem którejś z paczek; 0 — jeśli przejście odbywa się bez przesuwania paczek. Odpowiedź uzyskamy obliczając długość najtańszej ścieżki od stanu początkowego do jednego ze stanów końcowych.

Niestety, algorytm ten jest niepraktyczny. Graf stanów ma bowiem N^2 wierzchołków, gdzie N jest liczbą pól w magazynie. Jest to nieakceptowalne, gdy tak jak w naszym zadaniu, N może się równać 10000. Graf G_S możemy łatwo zmodyfikować tak, by miał jedynie $O(N)$ wierzchołków. Odbędzie się to jednak kosztem bardziej skomplikowanego wyznaczania krawędzi. Zauważmy bowiem, że z naszego punktu widzenia istotne są tylko te stany, w których pole magazyniera sąsiaduje z polem paczki, a takich pól jest mniej niż $4N$. Tylko z tych stanów wychodzą krawędzie o wadze 1, a więc odpowiadające przesunięciu paczki. Pozostałe krawędzie, o wadze 0, będą łączyły pary stanów różniące się jedynie pozycją magazyniera i będą odpowiadały „obejściu" paczki przez magazyniera (wykonanym jedynie po to, by zmienić kierunek pchania).

Krawędzie takie można wyznaczyć różnymi metodami. Poniżej zasygnalizujemy dwie z nich. Ponieważ interesują nas teraz ruchy magazyniera, w których nie przesuwa on paczki, wygodnie jest posłużyć się innym grafem, nazwijmy go G_M, którego wierzchołkami są pola magazynu nie zajęte przez skrzynie, a krawędzie łączą wierzchołki odpowiadające sąsiadującym polom. Zwracamy uwagę, że rozważany w tej chwili graf G_M jest nową konstrukcją i nie należy go mylić z grafem stanów G_S.

Metoda pierwsza. Dla każdej pary stanów u, v z grafu G_S różniących się jedynie pozycjami magazyniera sprawdzamy, czy w G_M istnieje ścieżka łącząca te pozycje i nie przechodząca przez pole zajęte przez paczkę.

Metodę tę można sprowadzić do zastosowania procedury przechodzenia grafu G_M w głąb: najpierw usuwamy z G_M wierzchołek (wraz z incydentnymi krawędziami), na którym stoi paczka, a następnie rozpoczynamy przechodzenie G_M od pola zajętego przez magazyniera w stanie u. Krawędź między stanami istnieje, jeśli uda nam się dojść do pola zajętego przez magazyniera w stanie v. Co prawda procedura przechodzenia grafu w głąb działa w czasie liniowym, ale konieczność wykonania jej dla $O(N)$ par stanów prowadzi do czasu kwadratowego.

Bardziej uważna analiza pozwala wyznaczyć krawędzie G_S w znacznie krótszym czasie. Zauważamy, że parę stanów, w których paczka zajmuje to samo pole, należy połączyć krawędzią wtedy i tylko wtedy, gdy w G_M istnieją co najmniej dwie rozłączne ścieżki pomiędzy polami odpowiadającymi pozycjom magazyniera. Jedna ścieżka zawsze istnieje — jest to ścieżka długości 2, wiodąca jedynie poprzez pole zajęte przez paczkę. Jeśli więc ma być możliwość obejścia paczki, musi istnieć w G_M druga ścieżka. Grafy, w których zachodzi taka własność, nazywamy dwuspójnymi.

Definicja 1: Graf $G' = (V', E')$ jest *dwuspójny*, jeśli pomiędzy każdą parą jego wierzchołków istnieją co najmniej dwie łaczące je ścieżki, które poza końcami nie mają żadnych wspólnych wierzchołków.

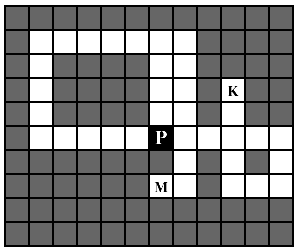
Metoda druga. Rozbijamy graf G_M na *dwuspójne składowe*, tj. na maksymalne podgrafy, z których każdy jest grafem dwuspójnym. Parę stanów u, v z grafu G_S, różniących się jedynie pozycjami magazyniera, łączymy krawędzią wtedy i tylko wtedy, gdy pola zajęte przez magazyniera w u i v należą do jednej dwuspójnej składowej z G_M.

Algorytm wyznaczający dwuspójne składowe polega na sprytnym wykorzystaniu algorytmu przechodzenia grafu wgłąb i działa w czasie liniowym. Nie będziemy go omawiać w tym miejscu, a zainteresowanego czytelnika odsyłamy do [12].

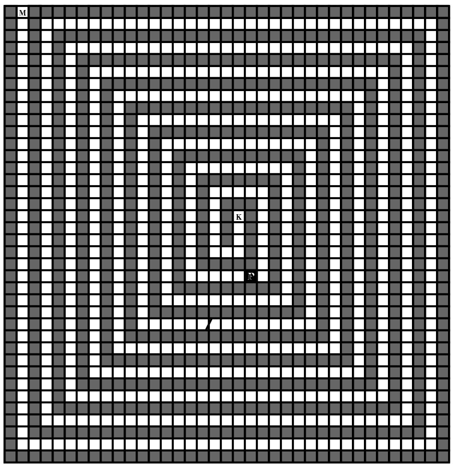
TESTY

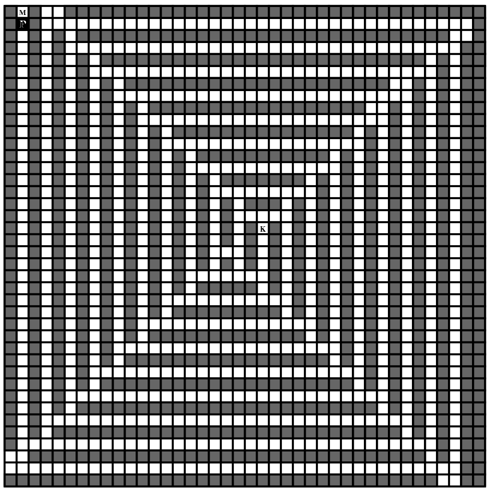
Do oceny rozwiązań użyto 18 testów MAG0.IN–MAG17.IN. Test MAG0.IN (rysunek 1) jest testem z treści zadania. Pozostałe można podzielić na kategorie w zależności od kształtu magazynu. Poniżej ilustrujemy kilka z nich.

Rys. 1 Test MAG0.IN. Dla przejrzystości pola zajęte przez skrzynie zamalowane są kolorem szarym, pole zajęte przez paczkę — kolorem czarnym, a pola wolne — kolorem białym.

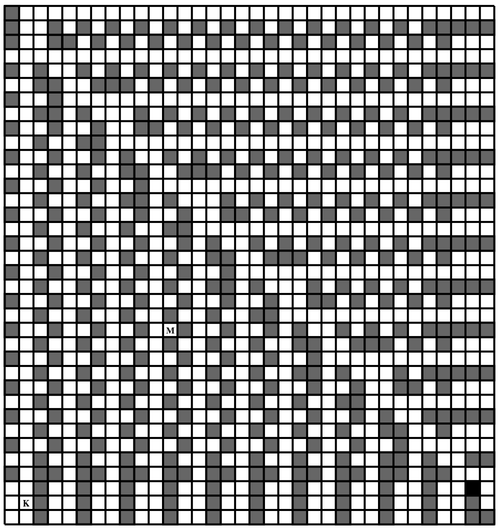


Rys. 2 Testy „spiralne”. Tego typu testami są MAG2.IN-MAG5.IN. Podobnymi testami są też MAG14.IN i MAG15.IN — w nich jednak korytarze mają podwójną szerokość.

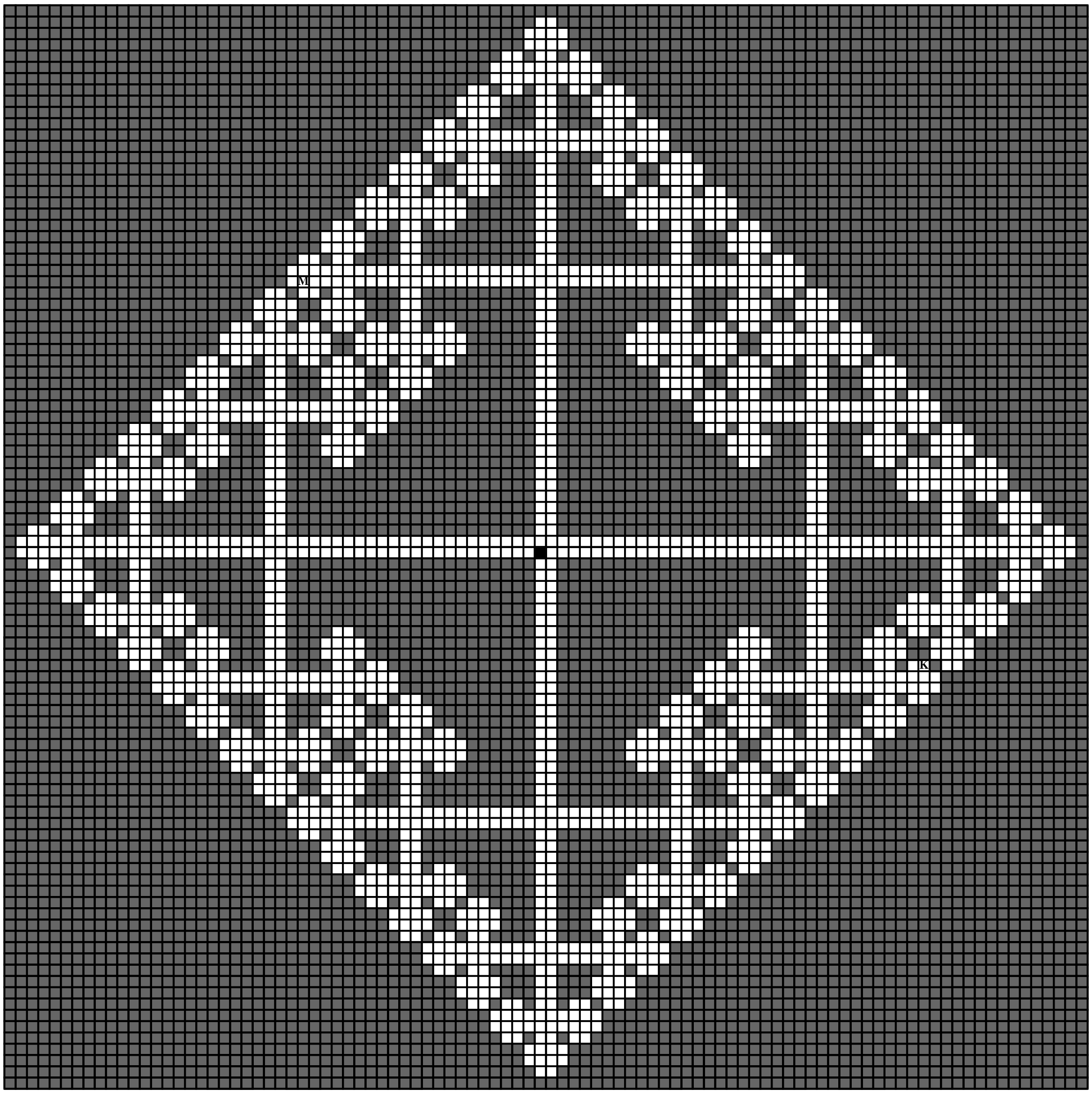




Rys. 3 Testy wymuszające zawracanie. Zwróć uwagę, że magazynier będzie musiał przepychać paczkę do wszystkich komór znajdujących się na prawym i dolnym obrzeżu magazynu, ponieważ tylko tam może obejść paczkę i zmienić kierunek pchania. Tego typu testami są MAG16.IN i MAG17.IN.



Rys. 4 Test „fraktalny”. Tego typu testami są MAG8.IN i MAG9.IN.



Marcin Sawicki
autor zadania, opracowanie

Marek Pawlicki
program wzorcowy

Mapa

Po nowym podziale administracyjnym Bajtocji przygotowywana jest mapa demograficzna kraju. Z powodów technicznych do kolorowania mapy można użyć tylko kilku kolorów. Mapę należy pokolorować w taki sposób, żeby gminy o zbliżonym zaludnieniu (tj. liczbie mieszkańców) były pokolorowane tym samym kolorem. Dla danego koloru k niech $A(k)$ będzie taką liczbą, że wśród gmin o kolorze k jest tyle samo gmin o zaludnieniu większym lub równym $A(k)$, co gmin o zaludnieniu mniejszym lub równym $A(k)$. Błędem pokolorowania *gminy pokolorowanej kolorem k nazywamy wartość bezwzględną różnicy liczby $A(k)$ i zaludnienia gminy.* Błędem sumarycznym *nazywamy sumę wszystkich błędów pokolorowania. Jak pokolorować mapę, żeby błąd sumaryczny był najmniejszy?*

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `MAP.IN` *zaludnienia gmin Bajtocji,*
- *obliczy minimalny błąd sumaryczny,*
- *zapisze wynik w pliku tekstowym* `MAP.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `MAP.IN` *znajduje się jedna liczba całkowita n równa liczbie gmin Bajtocji, $10 < n < 3000$. W drugim wierszu jest zapisana liczba m kolorów użyta do pokolorowania mapy, $2 \leqslant m \leqslant 10$. W każdym z następnych n wierszy znajduje się po jednej nieujemnej liczbie całkowitej. Są to zaludnienia gmin Bajtocji. Populacja Bajtocji nie przekracza 2^{30}.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego `MAP.OUT` *jedną liczbę całkowitą będącą równą minimalnemu błędowi sumarycznemu, jaki można uzyskać przy kolorowaniu mapy.*

Przykład

Dla pliku wejściowego `MAP.IN`*:*

```
11
3
21
14
6
18
10
2
15
12
3
2
2
```

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

```
15
```

MODYFIKACJA TREŚCI ZADANIA

Podczas trwania zawodów definicję liczby $A(k)$ zmieniono w następujący sposób:
Liczba $A(k)$ to dowolna liczba rzeczywista taka, że:

- *co najmniej połowa spośród gmin pokolorowanych kolorem k ma $A(k)$ lub więcej mieszkańców,*
- *co najmniej połowa spośród gmin pokolorowanych kolorem k ma $A(k)$ lub mniej mieszkańców,*

ROZWIĄZANIE

Liczba $A(k)$, zdefiniowana w treści zadania, jest *medianą* multizbioru* liczb mieszkańców gmin, oznaczonych na mapie kolorem k. Mediana multizbioru o parzystej liczbie elementów może nie być wyznaczona jednoznacznie. Mediana multizbioru liczb całkowitych nie musi nawet (choć może) być liczbą całkowitą. Zauważmy jednak, że wartość błędu sumarycznego jest dobrze określona mimo to. Po prostu nie zależy od tego, którą z możliwych wartości mediany wybierzemy do obliczeń (Czytelnik zechce to sprawdzić!).

Oznaczmy przez $L[1..n]$ tablicę liczb mieszkańców poszczególnych gmin. Załóżmy, że wartości te są posortowane niemalejąco. Łatwo zauważyć, że minimalny błąd sumaryczny jest osiągany dla takiego pokolorowania mapy, w którym każdy z kolorów jest użyty do pomalowania gmin ze spójnego fragmentu tablicy L. Dlatego wprowadzimy dalsze oznaczenia: niech $W[g,j]$ będzie sumą błędów pokolorowania gmin $g, g+1, \ldots, j$ przy założeniu, że wszystkie one (i tylko one) zostały pokolorowane tym samym kolorem. Czyli $W[g,j] = |L[g]-A| + |L[g+1]-A| + \cdots + |L[j]-A|$, gdzie A jest medianą liczb $L[g], \ldots, L[j]$. Możemy przyjąć np. $A = L\left[\left\lfloor \frac{g+j+1}{2} \right\rfloor\right]$. Przyjmijmy na razie, że znamy wartości $W[g,j]$ dla wszystkich $1 \leqslant g \leqslant j \leqslant n$.

Zbliżamy się do rozwiązania naszego problemu metodą programowania dynamicznego. Niech $s[g,i]$ będzie sumą błędów optymalnego pokolorowania przedziału $g \ldots n$ przy pomocy i kolorów. Oznaczenie to ma sens tylko dla $g \leqslant n-i+1$, zaś dla $1 \leqslant g \leqslant m-i$ nie będzie dla nas użyteczne. Oczywiście $s[g,1] = W[g,n]$. Zachodzi także (co jest dla nas kluczowe) równość

$$s[g,i+1] = \min_{j=g,\ldots,n-i} \left(W[g,j] + s[j+1,i]\right).$$

W istocie, jeśli przedział $g \ldots n$ został pokolorowany optymalnie $i+1$ kolorami, a pierwszym kolorem pokolorowano gminy od g do j włącznie, to gminy od $j+1$ do n również muszą być pokolorowane optymalnie, tyle że za pomocą i kolorów. Przecież pozostawiając nie zmienione j, za to poprawiając pokolorowanie gmin od $j+1$-szej do n-tej, poprawilibyśmy pokolorowanie całego przedziału $g \ldots n$, wbrew założeniu, że było ono optymalne!

Potrzebujemy znaleźć wartość $s[1,m]$. Teraz już bardzo łatwo ułożyć algorytm, który to zrobi:

```
for g := 1 to n do
  s[g,1] := W[g, n]
for i := 2 to m do
  for g := m − i + 1 to n − i + 1 do begin
    s[g,i] := +∞
    for j := g to n − i + 1 do
      s[g,i] := min(s[g,i], W[g,j] + s[j + 1, i − 1])
  end
end
```

* Multizbiór (ang. *multiset, bag*) jest strukturą danych podobną do zbioru, z tym że elementy mogą w nim występować wielokronie, np. $\{1,3,3,5\}$, $\{1,1,3,5\}$.

Widoczne jest typowe dla programowania dynamicznego wykorzystanie wyników częściowych, czyli wartości z tablicy s, dla obliczenia kolejnych wyników częściowych oraz wyniku ostatecznego.

Algorytm ten działa oczywiście w czasie $O(mn^2)$ i potrzebuje $O(n)$ słów pamięci, gdyż nie musimy przechowywać całej tablicy s, a jedynie dwa ostatnie jej wiersze. Gdybyśmy potrzebowali również odtworzyć znalezione optymalne pokolorowanie, potrzebowalibyśmy dodatkowej tablicy o rozmiarze $m \times n$ do przechowywania tych wartości j, dla których w najbardziej wewnętrznej pętli zostało znalezione minimum.

Jeśli tablica L nie była na wejściu posortowana, to możemy ją posortować w czasie $O(n \log n)$, który oczywiście jest zaniedbywalny z punktu widzenia naszej analizy.

Pozostaje problem znalezienia wartości W. Naiwny algorytm wyznaczania każdego $W[g,j]$ z osobna wprost z definicji (jako odpowiedniej sumy) wymaga czasu $\Omega(n^3)$, co istotnie spowalnia nasz program. Okazuje się, że i tym razem w sukurs przychodzi nam technika programowania dynamicznego: zachodzi zależność

$$W[g,j+1] = W[g,j] + L[j+1] - L\left[\left\lfloor \frac{g+j+1}{2} \right\rfloor\right],$$

która pozwala wyznaczyć całą tablicę W w czasie $O(n^2)$. Zamiast dowodzić tej równości formalnie, proponuję przyjrzeć się rysunkowi 1.

Nietrudno też spostrzec, że nie potrzebujemy trzymać wartości $W[g,j]$ w żadnej tablicy: jeżeli będziemy obliczać je w miarę potrzeby w wewnętrznej pętli algorytmu, to co prawda każdą z nich obliczymy $(m-1)$-kronie, lecz złożoność czasowa całego algorytmu na tym nie ucierpi (nadal będzie wynosić $O(mn^2)$), zaś złożoność pamięciowa pozostanie na poziomie $O(n)$. Przy podanych ograniczeniach na n i m jest to istotne usprawnienie.

INNE ROZWIĄZANIA

Wspomniano już o rozwiązaniu o złożoności czasowej $\Omega(n^3)$, obliczającym W wprost z definicji. Taki algorytm potrzebuje również $\Omega(n^2)$ słów pamięci na przechowywanie tablicy W. Jeśli chcielibyśmy zaoszczędzić pamięć, to koszt wzrósłby do $\Omega(mn^3)$. Można też ułożyć algorytm przeglądający wszystkie możliwe podziały tablicy L na m bloków; działałby on w czasie $\Omega(n^{m-1})$. Jakość poszczególnych algorytmów łatwo ocenić wiedząc, że w Polsce jest około 2500 gmin, a do kolorowania map demograficznych używa się zazwyczaj 7 kolorów.

INNY WARIANT ZADANIA

W praktyce kartograficznej nie minimalizuje się sumy odległości danych od ich mediany, lecz od ich średniej arytmetycznej. Oczywiście wymaga to użycia arytmetyki zmiennopozycyjnej. Zwróćmy uwagę, że dla rozwiązania tego zadania nie musimy modyfikować głównej części algorytmu, a jedynie sposób obliczania wartości $W[g,j]$.

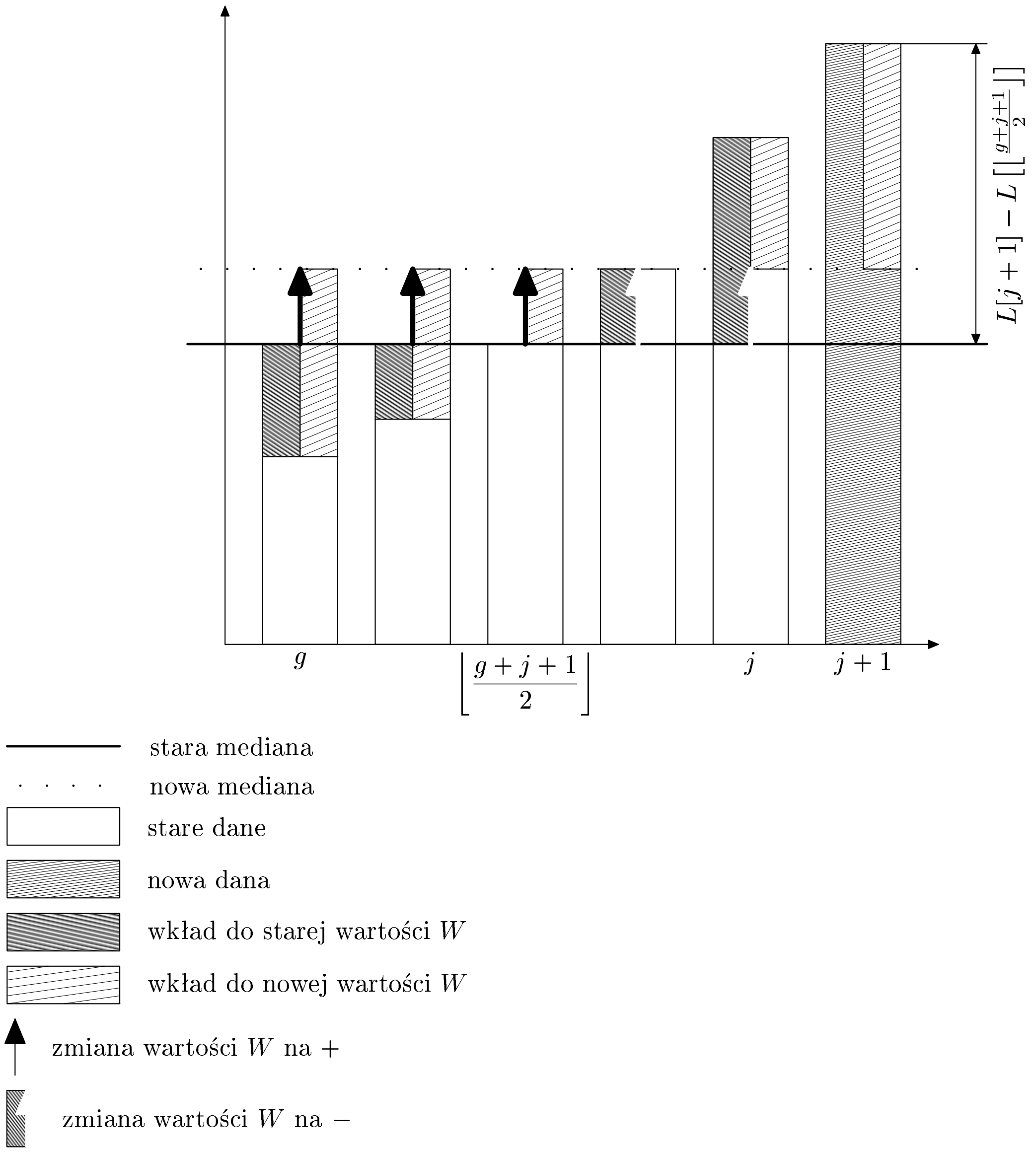
Tym razem również można wyznaczyć wszystkie wartości $W[g,j]$ w łącznym czasie $O(n^2)$ Co więcej tak samo jak poprzednio nie potrzebujemy trzymać ich wszystkich w tablicy. Jednak obliczenie $W[g,j+1]$ na podstawie $W[g,j]$ nie jest, jak sądzę, możliwe w czasie stałym (najkrócej mówiąc, jeśli $L[j+1]$ jest stosunkowo duże, to wartość średnia może przeskoczyć kilka spośród liczb L na raz). Daje się to natomiast zrobić w zamortyzowanym czasie stałym.* Znalezienie odpowiedniego algorytmu może być ciekawym ćwiczeniem.

TESTY

Do oceny rozwiązań użyto 10-ciu testów MAP0.IN–MAP9.IN. Test MAP0.IN był testem z treści zadania. Testy MAP1.IN–MAP5.IN sprawdzały poprawność programu. Był wśród nich zestaw danych o minimalnym rozmiarze, zestaw, w którym wszystkie gminy miały po 1 mieszkańcu, dwa proste testy losowe i jeden,

* Definicję i omówienie pojęcia kosztu zamortyzowanego znajdzie Czytelnik w książce [12], rozdział 18.

Rys. 1 Indukcyjne obliczanie wartości $W[g, j]$.



w którym populacja całej Bajtocji wynosiła $2^{30} - 2$. Następne cztery testy były losowe i coraz większe — zaprojektowane tak, by algorytm działający w czasie $\Omega(n^{m-1})$ przechodził co najwyżej pierwszy z nich, a działający w czasie $\Omega(n^3)$ — co najwyżej dwa pierwsze.

Grzegorz Jakacki
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Ołtarze

Według chińskich wierzeń ludowych złe duchy mogą poruszać się tylko po linii prostej. Ma to istotne znaczenie przy budowie świątyń. Świątynie są budowane na planach prostokątów o bokach równoległych do kierunków północ-południe oraz wschód-zachód. Żadne dwa z tych prostokątów nie mają punktów wspólnych. Po środku jednej z czterech ścian jest wejście, którego szerokość jest równa połowie długości tej ściany. W centrum świątyni (na przecięciu przekątnych prostokąta) znajduje się ołtarz. Jeśli znajdzie się tam zły duch, świątynia zostanie zhańbiona. Tak może się zdarzyć, jeśli istnieje półprosta (w płaszczyźnie równoległej do powierzchni terenu), która biegnie od ołtarza w centrum świątyni przez otwór wejściowy aż do nieskończoności, nie przecinając i nie dotykając po drodze żadnej ściany, tej lub innej świątyni.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `OLT.IN` *opisy świątyń,*
- *sprawdzi, które świątynie mogą zostać zhańbione,*
- *zapisze ich numery w pliku tekstowyn* `OLT.OUT`.

Wejście

W pierwszym wierszu pliku wejściowego `OLT.IN` *jest zapisana jedna liczba naturalna* n, $1 \leqslant n \leqslant 1000$, *będąca liczbą świątyń.*

W każdym z kolejnych n *wierszy znajduje się opis jednej świątyni (w* i*-tym z tych wierszy opis świątyni numer* i*). Opis świątyni składa się z czterech nieujemnych liczb całkowitych, nie większych niż 8000, oraz jednej litery E, W, S lub N. Pierwsze dwie liczby, to współrzędne północno-zachodniego narożnika świątyni, a dwie następne, to współrzędne przeciwległego, południowo-wschodniego narożnika. Określając współrzędne punktu podajemy najpierw jego długość geograficzną (która rośnie z zachodu na wschód), a następnie — szerokość geograficzną, która rośnie z południa na północ. Piąty element opisu wskazuje ścianę, na której znajduje się wejście do świątyni (E — wschodnią, W — zachodnią, S — południową, N — północną). Kolejne elementy opisu świątyni są pooddzielane pojedynczymi odstępami.*

Wyjście

W kolejnych wierszach pliku tekstowego `OLT.IN` *Twój program powinien zapisać w porządku rosnącym numery świątyń, które mogą zostać zhańbione przez złego ducha, każdy numer w osobnym wierszu. Jeżeli takich świątyń nie ma, to w pliku* `OLT.OUT` *należy zapisać jedno słowo* `BRAK`.

Przykład

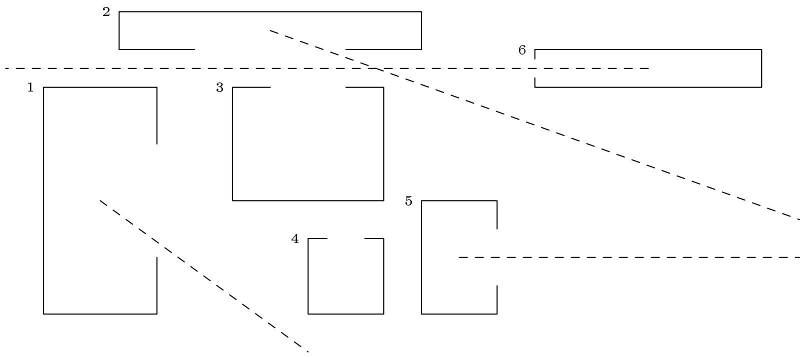
Dla pliku wejściowego `OLT.IN`*:*

```
6
1 7 4 1 E
3 9 11 8 S
6 7 10 4 N
8 3 10 1 N
11 4 13 1 E
14 8 20 7 W
```

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

```
1
2
5
6
```

Na rysunku pokazano układ świątyń opisany w pliku `OLT.IN`*. Linie przerywane pokazują możliwe drogi inwazji złych duchów.*



ROZWIĄZANIE

Zadanie reprezentuje dziedzinę leżącą na pograniczu algorytmiki i geometrii, tzw. geometrię obliczeniową. Jednym z bodźców, który przyczynił się do jej rozwoju, było pojawienie się problemów „z życia wziętych", związanych m.in. z zagadnieniami transportowymi, analizą obrazu, czy produkcją układów scalonych wysokiej skali integracji. Na taki właśnie z życia wzięty problem natrafiłem w Chinach. Świątynie występujące w zadaniu są bardzo silnym uproszczeniem rzeczywistości. Chińska świątynia przypomina park lub ogród, w którym znajduje się kilka pawilonów mieszczących przedmioty kultu. Świątynia posiada często oś symetrii, na której znajdują się główne zabudowania oraz brama. W poprzek bramy, na terenie świątyni lub poza nią, znajduje się zazwyczaj specjalny mur. Złe duchy i wszelkie „wpływy" poruszają się bowiem, według Chińczyków, wyłącznie po linii prostej. Ściana pełni przeto rolę ekranu zabezpieczającego świątynie przed wpływami świata zewnętrznego. Z podobnych przyczyn do wielu domostw wchodzi się przez wysoki próg lub zza węgła.

Przy wyznaczaniu lokalizacji nowej świątyni Chińczycy posługują się starożytną sztuką *Feng Shui* łączącą elementy wróżbiarstwa, radiestezji i planowania przestrzennego. Nam, nie mającym pojęcia o tej skompilkowanej metodzie, pozostaje rozwiązanie zadania przy pomocy geometrii obliczeniowej.

Skupmy się na tym, w jaki sposób sprawdzić, czy pewna wybrana świątynia S może zostać zhańbiona. Wyobraźmy sobie, że stoimy na ołtarzu tej świątyni ze wzrokiem utkwionym w lewej framudze otworu wejściowego (patrząc z wnętrza świątyni). Następnie zaczynamy powoli obracać się zgodnie z ruchem wskazówek zegara (w prawo). Linia naszego wzroku prowadzi ze świątyni przez otwór drzwiowy i przesuwa się po fragmentach innych świątyń. Natrafienie wzrokiem na prześwit pomiędzy zabudowaniami oznacza odnalezienie potencjalnej drogi inwazji złego ducha.

Niech O oznacza punkt położenia ołtarza w świątyni S. Każdą półprostą o wierzchołku O nazywać będziemy *drogą inwazji*. Mówimy, że droga inwazji jest *zabezpieczona przez świątynię T*, jeżeli ma punkt wspólny ze ścianą świątyni T. Zbiór wszystkich dróg inwazji zabezpieczanych przez świątynię T nazywamy *obszarem zasłanianym przez świątynię T*. *Polem widzenia świątyni S* nazywamy zbiór wszystkich dróg inwazji nie zabezpieczanych przez tę świątynię.

Każdy obszar zabezpieczany przez świątynię jest jednoznacznie wyznaczany przez dwie skrajne drogi inwazji. Jeżeli przyjmiemy uporządkowanie dróg inwazji zgodnie z ruchem wskazówek zegara, to wygodnie pierwszą z tych dróg nazywać *lewym skrajem*, natomiast drugą — *prawym skrajem*.

Niech X będzie zbiorem wszystkich obszarów zabezpieczanych przez świątynie. **Bez szkody możemy ograniczyć się do niepustych obszarów będących przecięciami obszarów z X z polem widzenia świątyni S.** Dalej tylko te obszary będziemy nazywać obszarami zasłanianymi.

Algorytm sprawdzający, czy świątynia S może zostać zhańbiona wygląda następująco:

(1) Dla każdej świątyni różnej od S wyznaczamy lewy i prawy skraj obszaru, który ona zabezpiecza i zapamiętujemy je, jeżeli obszar ma niepuste przecięcie z polem widzenia świątyni S.

(2) Sortujemy skraje zgodnie z porządkiem wyznaczonym przez ruch wskazówek zegara.

(3) Sprawdzamy, czy pierwszy i ostatni skraj pokrywa się odpowiednio z lewym i prawym skrajem pola widzenia (jeśli nie, to istnieje niezabezpieczona droga inwazji).

(4) Ustawiamy licznik na zero i przeglądamy kolejne skraje. Natrafiając na lewy skraj zwiększamy licznik, natrafiając na prawy — zmniejszamy. Napotkawszy sytuację, w której wartość licznika wynosi zero, a kolejny skraj ma kierunek różny od ostatniego skraju, kończymy sprawdzanie (wykryliśmy niezabezpieczoną drogę inwazji).

(5) Jeżeli sprawdzanie nie zakończyło się w poprzednich punktach, to każda droga inwazji jest zabezpieczona.

Optymalna implementacja algorytmu ma złożoność czasową $O(n \log n)$, pochodzącą od operacji sortowania. Ponieważ analogiczne sprawdzenie trzeba wykonać dla każdej świątyni osobno, ogólny koszt czasowy optymalnej implementacji wynosi $O(n^2 \log n)$. Umiejętna implementacja algorytmu pozwala uzyskać koszt pamięciowy $O(1)$.

Program wzorcowy do sortowania wykorzystuje hybrydę algorytmu sortowania przez kopcowanie (Heapsort) oraz przez wstawianie (Insertionsort). Z tego powodu pesymistyczna złożoność dla tej implementacji wynosi $O(n^3)$, jednak dla losowych danych oczekiwany czas pracy algorytmu wynosi $O(n^2 \log n)$. Podobnie, pesymistyczny koszt pamięciowy wynosi $O(n)$, jednak dla danych losowych oczekiwany koszt pamięciowy jest rzędu $\log n$ (co w zasadzie i tak nie ma znaczenia ze względu na niskie ograniczenie n).

Kilku słów wymaga jeszcze omówienie szczegółów związanych z reprezentacją półprostych oraz sprawdzaniem ich położenia. Ponieważ operujemy wyłącznie na półprostych o wspólnym wierzchołku, półprostą można wygodnie reprezentować przez dowolny leżący na niej wektor (pozwala to także uniknąć obliczeń zmiennopozycyjnych). Do ustalenie kolejności półprostych względem porządku wyznaczanego przez kierunek ruchu wskazówek zegara można wykorzystać iloczyn wektorowy (zob. [12]).

INNE ROZWIĄZANIA

Zauważmy, że jeżeli nie musielibyśmy każdorazowo sortować skrajów, to można by przyspieszyć podane rozwiązanie. W istocie istnieje algorytm o złożoności czasowej $O(n^2)$, który dla każdego z n punktów na płaszczyźnie wyznacza uporządkowanie pozostałych $n-1$ punktów w porządku wyznaczonym przez ruch wskazówek zegara. Zastosowanie go do wstępnego posortowania narożników świątyń pozwoliłoby zredukować całkowity koszt czasowy rozwiązania do $O(n^2)$. Niestety algorytm ten jest zbyt skomplikowany, by go tu opisywać, nie wspominając o perspektywach zakodowania go podczas 5-godzinnej sesji zawodów Olimpiady.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11-tu testów OLT0.IN–OLT10.IN.

- OLT0.IN — test z treści zadania;
- OLT1.IN — mały test poprawnościowy;
- OLT2.IN — test z odpowiedzią „BRAK”;
- OLT3.IN — mały test losowy;
- OLT4.IN–OLT10.in — duże testy wydajnościowe.

Wojciech Rytter **Marcin Mucha**
autor zadania, opracowanie program wzorcowy

Pierwotek abstrakcyjny

Kod genetyczny *pierwotka abstrakcyjnego (*Primitivus recurencis*) jest ciągiem liczb naturalnych* $K = (a_1, \ldots, a_n)$*.* Cechą *pierwotka nazywamy każdą parę liczb* (l, r)*, które występują kolejno w kodzie genetycznym, tzn. istnieje i takie, że* $l = a_i$*,* $r = a_{i+1}$*. U pierwotków nie występują cechy postaci* (p, p)*.*

Zadanie

Napisz program, który:

- *wczyta listę cech z pliku tekstowego* `PIE.IN`*,*
- *obliczy długość najkrótszego kodu genetycznego pierwotka zawierającego podane cechy,*
- *zapisze wynik w pliku tekstowym* `PIE.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `PIE.IN` *zapisana jest jedna dodatnia liczba całkowita* n*. Jest to liczba różnych cech pierwotka. W każdym z kolejnych* n *wierszy znajduje się para liczb naturalnych* l *i* r *oddzielonych pojedynczym odstępem,* $1 \leqslant l \leqslant 1000$*,* $1 \leqslant r \leqslant 1000$*. Para* (l, r) *jest jedną z cech pierwotka. Cechy w pliku wejściowym nie powtarzają się.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `PIE.OUT` *dokładnie jedną liczbę całkowitą, równą długości najkrótszego kodu genetycznego pierwotka zawierającego cechy z pliku* `PIE.IN`*.*

Przykład

Dla pliku wejściowego `PIE.IN`*:*

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

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

```
15
```

Wszystkie cechy z pliku `PIE.IN` *są zawarte np. w następującym kodzie genetycznym:*

$$(8, 5, 1, 4, 2, 3, 9, 6, 4, 5, 7, 6, 2, 8, 6)$$

ROZWIĄZANIE

Zadanie ma charakter teoriografowy. Tworzymy graf skierowany G, którego wierzchołkami są liczby występujące w cechach pierwotka. Niech V oznacza zbiór tych liczb. W naszym zadaniu $V \subseteq \{1, \ldots, 1000\}$.

Każda cecha odpowiada jednej krawędzi. Niech E będzie zbiorem krawędzi grafu. Mamy zatem graf $G = (V, E)$, $|V| \leqslant 1000$, $|E| = m$.

Przez *ścieżkę* w grafie G rozumiemy taki ciąg krawędzi, że początek każdej kolejnej krawędzi jest końcem poprzedniej oraz żadna krawędź na ścieżce nie powtarza się (ścieżka może być cyklem). Mówimy, że zbiór ścieżek S *pokrywa* graf G, jeżeli każda krawędź grafu G należy do pewnej ścieżki ze zbioru S.

Obserwacja 1: Niech q będzie minimalną liczbą skierowanych, krawędziowo rozłącznych ścieżek, które pokrywają G. Wtedy rozwiązaniem zadania jest liczba $m + q$.

Zaczniemy od algorytmu A, który konstruuje najmniejszy zbiór ścieżek pokrywających graf G, a następnie składa te ścieżki w kod K, który jest najkrótszym kodem dla danego zestawu cech. Potem pokażemy, że można obliczyć liczbę q tych ścieżek bez faktycznego ich konstruowania (algorytm B).

Oznaczmy przez *path*(v) operację polegającą na znalezieniu nieprzedłużalnej ścieżki o początku w v i usunięciu jej krawędzi z grafu. Taką operację można wykonać startując z v i idąc dowolnie po grafie, jak długo się da, za każdym razem usuwając krawędź, po której przeszliśmy.

Niech $\gamma = (v_1, \ldots, v_{r-1}, v_r, v_1)$ będzie cyklem, a $\gamma' = ({v'}_1, \ldots, {v'}_{p-1}, {v'}_p)$ ścieżką o wspólnym wierzchołku u, $u = v_i = {v'}_j$, dla pewnych i, j, $1 \leqslant i \leqslant r$, $1 \leqslant j \leqslant p$. Operację wklejania cyklu γ w ścieżkę γ' definiujmy jako operację polegającą na zastąpieniu wierzchołka u na ścieżce γ' przez ścieżkę (cykl) $v_i, v_{i+1}, \ldots, v_r, v_1, \ldots, v_{i-1}, v_i$ i oznaczamy ją przez *paste*(γ,γ').

Oznaczmy przez $d^+(v)$, $d^-(v)$ liczby krawędzi, odpowiednio, wchodzących do i wychodzących z wierzchołka v. Liczby te nazywamy stopniem wejściowym i stopniem wyjściowym wierzchołka. Algorytm A konstruuje najkrótszy kod w sposób zachłanny i wygląda następująco:

```
function DługośćKoduA : integer;
begin
  S := ∅;
  while E ≠ ∅ do begin
    weźmy dowolny wierzchołek v taki, że d⁺(v) < d⁻(v);
    jeśli takiego wierzchołka nie ma, to weźmy dowolny wierzchołek taki,
    że d⁻(v) > 0;
    γ := path(v);
    if γ jest cyklem i ma wspólny wierzchołek z pewną ścieżką γ' ∈ S
    then
      paste(γ, γ');
    else
      S := S ∪ {γ}
  kod := kod powstały w wyniku sklejenia ścieżek* z S
  DługośćKoduA = |kod|
end
```

Rysunki 1 i 2 przedstawiają działanie algorytmu na przykładzie grafu odpowiadającego 14-tu cechom.

Algorytm A „liczy za dużo”, ponieważ nie interesują nas elementy zbioru S, a tylko ich liczba. Najsłabszą stroną algorytmu jest to, że musimy wczytać wszystkie cechy pierwotka do pamięci. Tak więc potrzebujemy pamięci rzędu m, a może się zdarzyć, że np. $|V| = 1000$, $m = 990000$.

Pokażemy teraz algorytm, w którym wystarczy pamięć rzędu liczby wierzchołków. Algorytm ten opiera się na analizie spójnych składowych grafu cech.

Słabo spójną składową (w skrócie: *składową*) grafu skierowanego nazywamy każdy maksymalny (w sensie zawierania) podzbiór jego wierzchołków taki, że między dowolnymi dwoma jego elementami istnieje ścieżka nieskierowana (zapominamy o orientacji krawędzi). Na rysunku 1 przedstawiono trzy spójne składowe przykładowego grafu.

Zdefiniujmy *stopień niezrównoważenia wierzchołka* v: $stn(v) = |d^+(v) - d^-(v)|$. Przez *stopień niezrównoważenia grafu* G (ozn. $stn(G)$) będziemy rozumieć sumę stopni niezrównoważenia jego wierzchołków. Mówimy, że graf jest zrównoważony gdy jego stopień niezrównoważenia wynosi zero.

* Tutaj ścieżki traktujemy jako ciągi wierzchołków

Obserwacja 2: Załóżmy, że graf G składa się z jednej spójnej składowej i $stn(G) \neq 0$. Wtedy w funkcji każdy obliczony cykl zostaje wklejony w ścieżkę, a każda ścieżka dodana do S zmniejsza stopień niezrównoważenia grafu o 2. Zatem po zakończeniu działania algorytmu $|S| = stn(G)/2$.

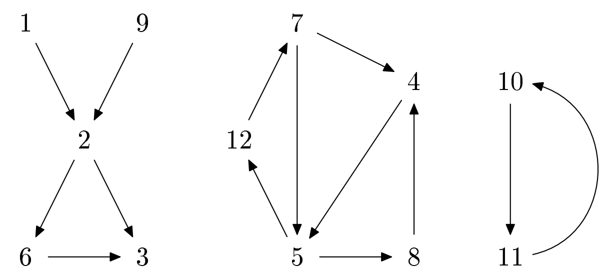
Z obserwacji tej wynika następujący wzór, w którym $G_1,\ G_2, \ldots G_r$ oznaczają składowe grafu G:

$$\text{dł. najkrótszego kodu} = \sum_{i=1}^{r} \max\{stn(G_i)/2,\ 1\} \tag{1}$$

W algorytmie wzorcowym liczymy $deficyt(v) = \max\{d^+(v) - d^-(v), 0\}$. Przez *deficyt grafu* rozumiemy sumę deficytów jego wierzchołków. Wtedy można zastąpić wzór (1) następującym równoważnym wzorem:

$$\text{dł. najkrótszego kodu} = \sum_{i=1}^{r} \max\{deficyt(G_i),\ 1\} \tag{2}$$

Rys. 1 Graf reprezentuje 14 cech oraz ścieżki i cykle skonstruowane w algorytmie zachłannego chodzenia.



P1: $1 \to 2 \to 6 \to 3$

P2: $9 \to 2 \to 3$

P3: $7 \to 4 \to 5 \to 8 \to 4$

C1: $7 \to 5 \to 12$

C2: $10 \quad 11$

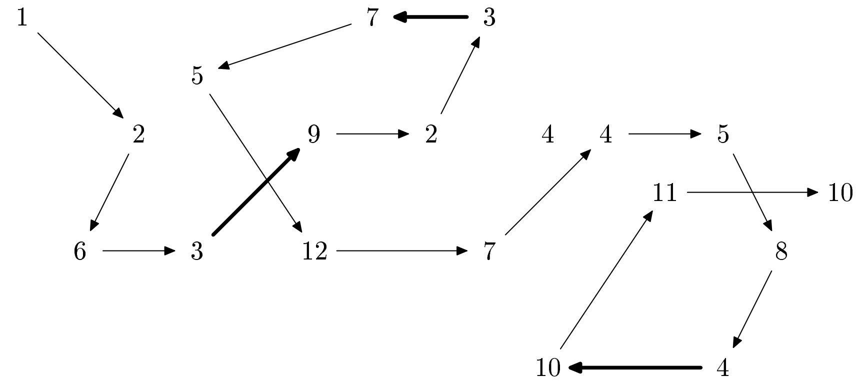
tablica stopni

i	1	2	3	4	5	6	7	8	9	10	11
$d^-[i]$	1	2	0	1	2	1	2	1	1	1	1
$d^+[i]$	0	2	2	2	2	1	1	1	0	1	1

Korzystając z powyższych wzorów konstruujemy algorytm B, który z pliku wczytuje każdą cechę tylko raz, ale nie zapamiętuje jej, a jedynie aktualizuje tablice stopni d^+ i d^- oraz informację o składowych grafu rozpiętego na dotychczas wczytanych krawędziach (cechach).

```
function DługośćKoduB : integer
begin
  for v := 1 to 1000 do begin
    d+[v] := 0;
    d-[v] := 0
  end
```

Rys. 2 Minimalny kod pierwotka abstrakcyjnego dla zbioru cech odpowiadającego grafowi z poprzedniego rysunku. Końcowy ciąg jest złożeniem ścieżek P_1, P_2, P_3 i cykli C_1, C_2. Specjalnie dodanymi połączeniami są $(3,9)$, $(3,7)$, $(4,10)$. Cykl C_1 został wklejony w ścieżkę P_3, natomiast cykl C_2 występuje samodzielnie. Krawędzie pogrubione zostały dodane w wyniku scalania ścieżek.




```
  { Graf nie zawiera żadnych krawędzi i składa się z 1000
    jednowierzchołkowych składowych. }
  while (nie wczytano wszystkich cech) do begin
    wczytaj kolejną cechę (u, v);
    d⁻[u] := d⁻[u] + 1; d⁺[v] := d⁺[v] + 1;
    if u i v należą do różnych składowych
    then połącz te składowe w jedną
  end
  DługośćKoduB := oblicz wynik korzystając ze wzoru (1) lub (2)
end
```

Żeby efektywnie zaimplementować powyższy algorytm musimy szybko umieć stwierdzać, do jakiej składowej należy dany wierzchołek i łączyć różne składowe w jedną. Jest to klasyczny problem sumowania zbiorów rozłącznych (ang. *find-union*). Różne efektywne rozwiązania tego problemu znajdzie czytelnik w książkach [9] oraz [12]. W programie wzorcowym zastosowano metodę łączenia wg rangi z kompresją ścieżek [12]. Tak zaimplementowany algorytm działa w czasie prawie liniowym ze względu na m i używa niewielkiej ilości pamięci (kilka tablic długości 1000).

TESTY

Do sprawdzenia rozwiązań zawodników użyto 12-tu testów PIE0.IN–PIE11.IN. Test PIE0.IN był testem z treści zadania. Testy PIE1.IN–PIE6.IN to testy poprawnościowe. Testy PIE7.IN–PIE11.IN to testy sprawdzające wydajność (czasową i pamięciową) rozwiązań.

Marcin Kubica
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Trójkolorowe drzewa binarne

Drzewo *składa się z wierzchołka, do którego podczepiono zero, jedno lub dwa poddrzewa, zwane dziećmi.* Specyfikacją *drzewa nazywamy ciąg cyfr. Jeżeli drzewo składa się z wierzchołka, do którego podczepiono:*

- *zero dzieci, to jego specyfikacja jest ciągiem jednoelementowym, którego jedynym elementem jest cyfra '0';*
- *jedno dziecko, to jego specyfikacja jest ciągiem rozpoczynającym się od cyfry '1'; po której następuje specyfikacja dziecka;*
- *dwoje dzieci, to jego specyfikacja jest ciągiem rozpoczynającym się od cyfry '2', po której następuje najpierw specyfikacja jednego, a potem drugiego dziecka.*

Każdy wierzchołek drzewa trzeba pomalować na czerwono, zielono lub niebiesko. Należy jednak trzymać się dwóch zasad:

- *wierzchołek i jego dziecko nie mogą być pomalowane na ten sam kolor,*
- *jeżeli wierzchołek ma dwoje dzieci, to muszą one być pomalowane różnymi kolorami.*

Ile wierzchołków można pomalować na zielono?

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `TRO.IN` *specyfikację drzewa,*
- *obliczy maksymalną i minimalną liczbę wierzchołków, które można pomalować na zielono,*
- *zapisze wyniki w pliku tekstowym* `TRO.OUT`.

Wejście

Pierwszy i jedyny wiersz pliku wejściowego `TRO.IN` *zawiera słowo o długości nie przekraczającej 10000 znaków, będące specyfikacją pewnego drzewa.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego `TRO.OUT` *dokładnie dwie liczby całkowite oddzielone pojedynczym odstępem, odpowiednio maksymalną i minimalną liczbą wierzchołków, które można pomalować na zielono.*

Przykład

Dla pliku wejściowego `TRO.IN`*:*
```
1122002010
```
poprawną odpowiedzią jest plik tekstowy `TRO.OUT`*:*
```
5 2
```

ROZWIĄZANIE

Drzewo, zgodnie ze swoją definicją, może zawierać w sobie poddrzewa. Zastanówmy się, jaka może być minimalna/maksymalna liczba wierzchołków danego koloru w drzewie w zależności od kształtu tego drzewa i możliwej liczby wierzchołków danego koloru w jego poddrzewach. W treści zadania jest mowa o kolorze zielonym. Zauważmy jednak, że wybór konkretnego koloru nie ma znaczenia dla wyniku —

wszystkie kolory są traktowane tak samo. Jeśli jednak wiemy, jakiego koloru jest korzeń drzewa, to minimalna/maksymalna liczba wierzchołków danego koloru zależy od tego, czy jest to taki sam kolor, jak kolor korzenia, czy też inny kolor. Możemy więc uogólnić problem i dla zadanego drzewa wyliczyć cztery liczby: minimalną/maksymalną liczbę wierzchołków, które można pokolorować danym kolorem, takim samym/innym niż kolor korzenia. Dla każdego wierzchołka v naszego drzewa wyznaczymy cztery liczby opisujące poddrzewo o korzeniu w tym wierzchołku: a_v, b_v, c_v, d_v, gdzie a_v to minimalna liczba wierzchołków o tym samym kolorze co v, b_v to maksymalna liczba wierzchołków o tym samym kolorze co v, c_v to minimalna liczba wierzchołków tego samego koloru, innego niż kolor v, d_v to maksymalna liczba wierzchołków tego samego koloru, innego niż kolor v. Liczby te możemy wyliczyć w następujący sposób:

- Załóżmy, że v jest liściem (tzn. nie ma dzieci). Wówczas oczywiście $a_v = b_v = 1$ i $c_v = d_v = 0$.
- Załóżmy, że v ma jedno dziecko (o korzeniu) v'. Kolor v' musi się różnić od koloru v. Stąd $a_v = c_{v'} + 1$ oraz $b_v = d_{v'} + 1$. Kolor, który różni się od koloru v może być albo kolorem v', albo trzecim kolorem — różnym od kolorów v i v'. Stąd $c_v = \min(a_{v'}, c_{v'})$ oraz $d_v = \max(b_{v'}, d_{v'})$.
- Załóżmy, że v ma dwoje dzieci (o korzeniach) v' i v''. Wówczas v, v' i v'' muszą mieć trzy różne kolory. Stąd $a_v = c_{v'} + c_{v''} + 1$, $b_v = d_{v'} + dv'' + 1$, $c_v = \min(a_{v'} + c_{v''}, c_{v'} + a_{v''})$ oraz $d_v = \max(b_{v'} + d_{v''}, d_{v'} + b_{v''})$.

Zgodnie z powyższymi wzorami możemy wyznaczać liczby a_v, b_v, c_v, d_v idąc w górę drzewa, począwszy od liści, aż do korzenia. Jeżeli v jest korzeniem całego drzewa, to wynikiem są liczby $\min(a_v, c_v)$ i $\max(b_v, d_v)$

Powyższy algorytm można zaimplementować na kilka sposobów. Najprostszy wydaje się program zawierający rekurencyjną procedurę, która wczytuje specyfikację drzewa i daje w wyniku liczby a_v, b_v, c_v i d_v. Format wejścia jest taki, że każde wywołanie takiej procedury wczytuje z pliku wejściowego opis odpowiedniego poddrzewa. Wadą takiego rozwiązania jest to, że dla drzew o dużej wysokości może nastąpić przepełnienie stosu.

Pamięć potrzebną na stos można zmniejszyć konstruując drzewo, w którego węzłach znajdują się wyliczane wartości. Wówczas procedura rekurencyjna wyliczająca wartości przechowywane w węzłach może mieć tylko jeden argument, nie mieć zmiennych lokalnych i zajmować istotnie mniej pamięci na stosie. Mimo to, dla danych postaci

$$\underbrace{1 \ldots 1}_{9\,999\text{razy}}\ 0$$

nastąpi przepełnienie stosu.

Problemy z wielkością stosu można rozwiązać konstruując własny stos na wyliczane wartości i przerabiając procedurę rekurencyjną na iteracyjną. Możemy też najpierw wczytać specyfikację drzewa do tablicy, a następnie wyliczyć wynik iteracyjnie, korzystając z pomocniczego stosu.

Potraktujmy 0 jako stałą reprezentującą czwórkę $\langle 1, 1, 0, 0\rangle$, a 1 i 2, odpowiednio, jako jedno- i dwuargumentową operację wyliczającą czwórkę liczb $\langle a_v, b_v, c_v, d_v\rangle$ dla wierzchołka na podstawie czwórek liczb opisujących jego dzieci. Zauważmy wówczas, że format danych wejściowych jest wyrażeniem zapisanym w notacji polskiej. (W notacji polskiej najpierw zapisujemy operator, a potem jego argumenty. Np. $* + 3\ 2 - 3\ 1$ to $(3+2)*(3-1)$.) Jeżeli więc odwrócimy zapis danych wejściowych, to uzyskamy wyrażenie zapisane w odwrotnej notacji polskiej (z odwróconą kolejnością argumentów 2-ki, co nie zmienia wyniku). (W odwrotnej notacji polskiej najpierw zapisujemy argumenty, a potem operator. Np. $3\ 2 + 3\ 1 - *$ to $(3 + 2) * (3 - 1)$.) Wartość takich wyrażeń możemy wyliczać przetwarzając od lewej do prawej kolejne symbole w następujący sposób:

- jeśli dany symbol jest stałą, to włóż na stos wartość reprezentowaną przez tę stałą,
- jeśli dany symbol jest operacją, to zastosuj ją do wartości zdjętych z wierzchołka stosu, po czym wynik włóż na stos.

Po przetworzeniu całego wyrażenia na stosie powinna znajdować się jedna wartość będąca wynikiem wyrażenia.

Rozwiązanie takie zastosowano w programie załączonym na dyskietce. Poniżej przedstawiamy główną procedurę wyliczającą liczby a_v, b_v, c_v i d_v dla korzenia całego drzewa. Korzysta ona z tablicy *data* zawierającej dane wejściowe długości n, oraz ze stosu zaimplementowanego jako jednokierunkowa lista rekordów.

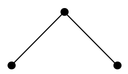
```
type
  stack = ↑stack_node;
  stack_node = record
    min_z, min_nz, max_z, max_nz : word;
    prev : stack;
  end;
procedure gen_ziel (var s : stack);
var
  i : word;
  p, q, r : stack;
begin
  for i := n downto 1 do begin
    new(p);
    case data[i] of
      '0' : begin
        p↑.min_nz := 0; p↑.min_z := 1;
        p↑.max_nz := 0; p↑.max_z := 1;
      end;
      '1' : begin
        q := pop(s);
        p↑.min_nz := min(q↑.min_nz, q↑.min_z);
        p↑.min_z := q↑.min_nz + 1;
        p↑.max_nz := max(q↑.max_nz, q↑.max_z);
        p↑.max_z := q↑.max_nz + 1;
        dispose(q);
      end;
      '2' : begin
        q := pop(s);
        r := pop(s);
        p↑.min_nz :=
          min(q↑.min_nz + r↑.min_z, q↑.min_z + r↑.min_nz);
        p↑.min_z := q↑.min_nz + r↑.min_nz + 1;
        p↑.max_nz :=
          max(q↑.max_nz + r↑.max_z, q↑.max_z + r↑.max_nz);
        p↑.max_z := q↑.max_nz + r↑.max_nz + 1;
        dispose(q); dispose(r)
      end
    end;
    push(s, p)
  end
end;
```

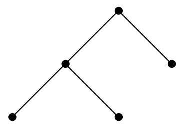
Zastanówmy się, czy nie można rozwiązać tego zadania znajdując pokolorowanie wierzchołków o najmniejszej i największej liczbie zielonych wierzchołków. Zauważmy, że w przypadku wierzchołków mających jedno lub dwoje dzieci, jeżeli kolor danego wierzchołka jest określony, to jego dzieci możemy pokolorować na dwa różne sposoby. Ponadto w całym drzewie liści jest o jeden więcej niż wierzchołków mających dwoje dzieci. Jeżeli więc n jest liczbą wierzchołków w drzewie, to przynajmniej $\lceil (n-1)/2 \rceil$ wierzchołków ma jedno lub dwoje dzieci. Stąd liczba możliwych pokolorowań jest wykładnicza ze względu na rozmiar drzewa. Tak więc programy sprawdzające wszystkie kolorowania drzewa dają wyniki w rozsądnym czasie jedynie dla bardzo małych drzew. Okazuje się też, że nie da się wyznaczyć kolorowania o maksymalnej lub minimalnej liczbie zielonych wierzchołków w sposób zachłanny. Tak więc kluczem do rozwiązania są zależności między liczbami a_v, b_v, c_v i d_v.

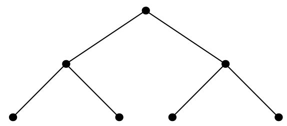
TESTY

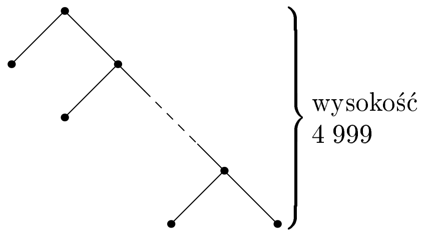
Do sprawdzenia rozwiązań zawodników użyto 13-tu testów TRO0.IN–TRO12.IN. Test TRO0.IN był testem z treści zadania. Testy TRO1.IN–TRO4.IN to małe testy poprawnościowe, a TRO5.IN i TRO6.IN to duże testy poprawnościowe. Zadaniem testów TRO1.IN–TRO4.IN było zbadanie ogólnej poprawności programu. Testy TRO5.IN i TRO6.IN badały, czy w programach nie występuje błąd przepełnienia stosu. Testy te przedstawiono na poniższych rysunkach. Testy TRO7.IN–TRO12.IN to testy losowe różnej wielkości. Ich zadaniem było zbadanie efektywności rozwiązań.

TRO1.IN

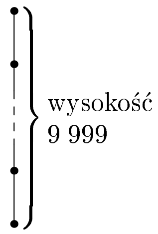

TRO2.IN


TRO3.IN


TRO4.IN



TRO5.IN



TRO6.IN

nr testu	**l. wierzch.**	**maks. l. w. ziel.**	**min. l. w. ziel.**
1	1	1	0
2	3	1	1
3	5	2	1
4	7	3	2
5	9 999	4 999	2 500
6	10 000	5 000	0
7	50	22	12
8	500	222	99
9	5 000	2 110	1 275
10	10 000	4 707	1 465
11	10 000	4 359	2 330
12	10 000	4 147	2 655

Grzegorz Jakacki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Woda

Na prostokątnej szachownicy składającej się z $n \times m$ pól ustawiono $n \cdot m$ prostopadłościanów — na każdym polu jeden prostopadłościan. Podstawa każdego prostopadłościanu pokrywa się z jednym polem szachownicy i ma powierzchnię jednego cala kwadratowego. Prostopadłościany na sąsiednich polach ściśle przylegają do siebie i nie tworzą żadnych szczelin. Na tę konstrukcję spadł ulewny deszcz. W niektórych miejscach utworzyły się zastoiska wody.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `WOD.IN` *rozmiary szachownicy oraz wysokości prostopadłościanów ustawionych na poszczególnych polach,*
- *obliczy maksymalną objętość wody, która po deszczu może pozostać w zastoiskach,*
- *zapisze wyniki w pliku tekstowym* `WOD.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `WOD.IN` *są zapisane dwie dodatnie liczby całkowite n i m, $1 \leqslant n \leqslant 100$, $1 \leqslant m \leqslant 100$. Są to rozmiary szachownicy. W każdym z kolejnych n wierszy znajduje się m liczb całkowitych z przedziału $[1..10000]$: i-ta liczba w j-tym wierszu jest wyrażoną w calach wysokością prostopadłościanu stojącego na przecięciu i-tej kolumny i j-tego wiersza szachownicy.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego `WOD.OUT` *jedną liczbę całkowitą równą maksymalnej objętości wody (wyrażoną w calach sześciennych), która może zebrać się w zastoiskach konstrukcji.*

Przykład

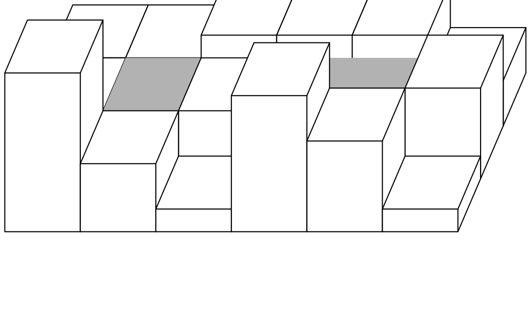
Dla pliku wejściowego `WOD.IN`*:*

```
3 6
3 3 4 4 4 2
3 1 3 2 1 4
7 3 1 6 4 1
```

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

```
5
```

Poniższy rysunek przedstawia szachownicę po deszczu.



ROZWIĄZANIE

Zacznijmy od ustalenia terminologii. *Wysokością* pola nazwiemy wysokość stojącego na nim prostopadłościanu. *Brzegiem* szachownicy nazwiemy zbiór tych pól, które mają boki wspólne z mniej niż czterema innymi polami.

Wyobraźmy sobie szachownicę jako pasmo górskie z kraterami, które powoli zalewa powódź. Woda podnosząc się zaczyna zalewać stoki gór położone przy brzegu szachownicy, ale do pewnego momentu kratery pozostają suche, chociaż ich dna znajdują się poniżej poziomu wody. Gdy wreszcie poziom wody zrównuje się z najniższym punktem korony jakiegoś krateru (punktem przelewowym), woda momentalnie zalewa krater. Zauważmy, że jeżeli woda przykryje szachownicę, po czym zupełnie opadnie, to poziom wody w każdym kraterze będzie dokładnie taki sam, jak w momencie jego zalania (zaniedbujemy parowanie). Oczywiście wszystko jedno, czy szachownicę zaleje powódź, czy deszcz.

Zaprojektujemy algorytm, który wykorzystuje powyższe obserwacje. Oznaczmy przez $h[p]$ wysokość pola p, a przez $l[p]$ *poziom wody* nad polem p po deszczu (tzn. odległość zwierciadła wody od szachownicy nad polem p). Jeżeli nad polem nie tworzy się zastoisko, to przyjmujemy $l[p] = h[p]$.

Zauważmy, że wartość $l[p]$ można interpretować jako poziom wody w momencie zalania. Dla pól, które leżą na stokach „odpływowych" jest to po prostu ich wysokość. Pola sortujemy niemalejąco względem wysokości i przeglądamy w tej kolejności (co odpowiada zalewaniu szachownicy przez podnoszącą się wodę). Jeżeli napotkamy pole „odpływowe" p, przez które woda może przelać się do krateru, „zalewamy" krater (tj. wszystkim polom p' krateru ustawiamy $l[p'] = h[p]$). Obszar krateru wyznaczamy na podstawie wysokości pól, jako maksymalny spójny obszar pól zawierający p, o wysokościach nie przekraczających $h[p]$. Zauważmy, że każde pole leżące na stoku „odpływowym" można traktować tak, jakby było „przelewowe". Co jednak robić, jeżeli przeglądając pola natkniemy się na pole leżące na dnie krateru? Nic. Wartość l dla tych pól zostanie wpisana w momencie zalania krateru. Algorytm przedstawia się następująco:

(1) Posortuj pola niemalejąco względem wysokości.

(2) Kolejno dla każdego pola p:

- jeżeli pole p należy do brzegu szachownicy lub sąsiaduje z polem p', dla którego obliczyliśmy już $l[p']$, to $l[p] := h[p]$ i „zalej krater" za pomocą algorytmu przeszukiwania grafu (np. DFS lub BFS*);
- w pozostałych przypadkach nie rób nic.

(3) Oblicz $\Sigma_p l[p] - h[p]$.

Zajmijmy się poprawnością opisanego algorytmu. *Ścieżką* nazwiemy taki ciąg pól, że każde dwa kolejne pola w tym ciągu mają wspólny bok, natomiast *wysokością ścieżki* — maksimum z wysokości jej pól.

Rozważmy ścieżki łączące pole p z brzegiem szachownicy. Spośród tych ścieżek wyróżnijmy taką, która ma minimalną wysokość i oznaczmy tę wysokość przez $l'[p]$. Widać, że:

- poziom wody na polu nie może być większy niż $l'[p]$, gdyż woda odpłynęłaby wyróżnioną ścieżką,
- poziom wody na polu nie może być mniejszy niż $l'[p]$, ponieważ wszystkie ścieżki, którymi woda mogłaby ujść z pola, mają wysokość przynajmniej $l'[p]$.

Tak więc poziom wody na polu p jest równy $l'[p]$. Jako ćwiczenie dla Czytelnika pozostawiamy udowodnienie faktu, że dla dowolnego pola p wielkość $l[p]$ obliczana przez opisany algorytm jest równa $l'[p]$.

Sortowanie pól można zrealizować w czasie $O(nm \log(nm))$, każdorazowe przeszukiwanie grafu można wykonać w czasie proporcjonalnym do rozmiaru zalewanego krateru. Łączny czas potrzebny na zalanie wszystkich kraterów nie przekroczy zatem $O(nm)$. Sumaryczny koszt czasowy algorytmu wynosi $O(nm \log(nm))$ i nie jest liniowy jedynie z powodu konieczności posortowania pól.

Czytelnik zechce sprawdzić, że ograniczenie algorytmu do przeglądania jedynie tych pól, dla których wartość l nie została obliczona, nie poprawia asymptotycznej złożoności czasowej.

* Opis algorytmów BFS i DFS można znaleźć w [12]

Złożoność pamięciowa algorytmu wynosi $O(nm)$. Sortowanie można zrealizować w pamięci $O(1)$ (tzn. używając tylko stałej liczby komórek pamięci ponad rozmiar danych wejściowych), natomiast przeszukiwanie grafu i składowanie wartości l wymaga pamięci rozmiaru $O(nm)$.

Implementacja wzorcowa korzysta z algorytmu sortowania przez kopcowanie (Heapsort) oraz algorytmu przeszukiwania grafu wszerz (BFS).

INNE ROZWIĄZANIA

Narzucającym się rozwiązaniem jest symulacja spływu wody. Oznaczmy przez h_{max} maksimum z wysokości pól szachownicy. Wiadomo, że dla dowolnego pola p zachodzi $h[p] \leqslant l'[p] \leqslant h_{max}$. Można skonstruować algorytm, który będzie wyznaczał wartości l' dla wszystkich pól metodą kolejnych przybliżeń. Dla każdego pola przechowywać będziemy wartość $w[p]$, początkowo równą h_{max}, która w miarę postępu obliczeń będzie „poprawiana":

- jeśli p leży na brzegu szachownicy, to $w[p] := h[p]$;
- jeśli pole ma sąsiada p' takiego, że $w[p'] < w[p]$, to $w[p] := max(h[p], w[p'])$.

Algorytm polega na „poprawianiu" wartości w dopóty, dopóki istnieje pole p, dla którego można „poprawić" wartość $w[p]$.

Jeżeli będziemy za każdym razem przeglądać wszystkie wierzchołki, to algorytm może wymagać $O(nm)$ „poprawek" dla całej planszy. W takim przypadku czas działania algorytmu wyniósłby $O((nm)^2)$.

TESTY

Do sprawdzania rozwiązań zawodników użyto 12-tu testów WOD0.IN–WOD12.IN:

- WOD0.IN — test z treści zadania;
- WOD1.IN — jeden wiersz;
- WOD2.IN — jedna kolumna;
- WOD3.IN — spirala na planszy 10×10;
- WOD4.IN — spirala na planszy 40×40;
- WOD5.IN — spirala na planszy 100×100;
- WOD6.IN — mały test poprawnościowy;
- WOD7.IN — maksymalny test poprawnościowy;
- WOD8.IN–WOD12.IN — testy losowe.