Zadanie: JUR
Juror

ONTAK 2013, dzień 3. Plik źródłowy `jur.*` Dostępna pamięć: 128 MB. *09.08.2013*

W tym zadaniu wcielasz się w jurora Olimpiady Informatycznej. Pracujesz właśnie nad przygotowaniem testów do jednego z zadań. Znalazłaś/znalazłeś wadliwe rozwiązanie i teraz musisz przygotować testy, których to rozwiązanie nie przejdzie. Niniejsze zadanie składa się z trzech podzadań. Twoim zadaniem jest napisanie *jednego* programu, który na podstawie danych wejściowych zorientuje się, z którym podzadaniem ma do czynienia, i znajdzie rozwiązanie dla tego podzadania.

W dziale Pliki znajdziesz paczkę z załącznikiem do tego zadania. Znajdują się tam kody źródłowe, o których mowa w dalszej części.

Uwaga. Za to zadanie można zdobyć w sumie 150 punktów.

Podzadanie 1. (50 punktów)

W zadaniu dany jest ciąg n liczb całkowitych, który należy posortować. W pliku `quicksort.cpp` znajduje się rozwiązanie, które implementuje algorytm Quicksort. To rozwiązanie, z powodu użytego sposobu wyboru elementu dzielącego, może działać wolno. Twoim zadaniem jest napisanie programu, który generuje takie testy, dla których łączna liczba iteracji dwóch wewnętrznych pętli **while** w funkcji `quicksort` będzie równa *co najmniej* $50n$.

Wejście

W pierwszym i jedynym wierszu wejścia znajduje się słowo `sortowanie`, po którym następuje liczba n ($200 \leq n \leq 100\,000$).

Wyjście

Twój program powinien wypisać dane wejściowe dla programu `quicksort.cpp` w następującym formacie. W pierwszym wierszu wyjścia powinna znaleźć się liczba n wczytana z wejścia. W drugim wierszu powinien znaleźć się ciąg n liczb całkowitych a_i ($1 \leq a_i \leq 10^9$).

Podzadanie 2. (50 punktów)

W zadaniu dana jest kwadratowa plansza o wymiarach $n \times n$ złożona z n^2 kwadratowych pól. Po planszy porusza się pionek. W każdym ruchu pionek może przesunąć się na jedno z czterech pól sąsiadujących z polem, na którym aktualnie się znajduje. Niektóre pola planszy są *zabronione* i pionek nie może się nigdy na nich znaleźć. Pozostałe pola nazywamy polami *dostępnymi*. Celem zadania jest obliczenie, dla każdego pola dostępnego, w ilu ruchach pionek może się na nie dostać. Poprawne* rozwiązanie tego zadania znajduje się w pliku `bfs_poprawny.cpp`. Zawiera ono implementację algorytmu BFS na podstawie szablonu `queue` z STL.

Łatwo jednak popełnić pewien (nieoczywisty) błąd i w rozwiązaniu użyć takiej implementacji kolejki, która zakłada, że każdym momencie w kolejce będzie co najwyżej 4000 elementów. W tym podzadaniu Twój program powinien wypisać test, dla którego powyższa procedura w pewnym momencie w trakcie wykonania będzie przechowywać w kolejce *więcej* niż 4000 elementów.

Wejście

W pierwszym i jedynym wierszu wejścia znajduje się słowo `bfs`, po którym następuje liczba n ($300 \leq n \leq 2000$).

Wyjście

Twój program powinien wypisać dane wejściowe dla programu `bfs_poprawny.cpp` w następującym formacie. Pierwszy wiersz powinien zawierać liczbę n wczytaną z wejścia, a następnie dwie liczby p_x i p_y ($1 \leq p_x, p_y \leq n$). Oznaczają one, że pionek startuje na polu w wierszu p_x i kolumnie p_y.

*Mamy nadzieję...

Dalej powinien znajdować się opis planszy w postaci n wierszy po n znaków `#` i/lub `.` — znaki te oznaczają odpowiednio pole zabronione i pole dostępne. Pole startowe musi być polem dostępnym.

Dla podanej planszy, w programie `bfs_poprawny.cpp` do kolejki powinno być wstawione więcej niż 4000 elementów.

Podzadanie 3. (50 punktów)

W zadaniu należy napisać algorytm znajdujący najkrótsze ścieżki w grafie nieskierowanym, w którym krawędzie mają długości wyrażone dodatnimi liczbami całkowitymi. Graf składa się z n wierzchołków, które numerujemy $1, \ldots, n$. Celem jest znalezienie odległości od wierzchołka 1 do każdego wierzchołka grafu. Poprawne rozwiązanie znaleźć można w pliku `dijkstra_poprawna.cpp`. Implementuje ono (nieco zmodyfikowany) algorytm Dijkstry przy użyciu `priority_queue`, czyli kolejki priorytetowej z STL.

Krótki opis modyfikacji. W książkowej wersji algorytmu Dijkstry dla każdego nieodwiedzonego wierzchołka, do którego znaleźliśmy już pewną ścieżkę, pamiętamy długość tej ścieżki w kolejce priorytetowej i tę wartość aktualizujemy. Modyfikacja w załączonym kodzie polega na tym, że w sytuacji, gdy znajdziemy nową, krótszą ścieżkę do danego wierzchołka, po prostu dodajemy nowy element do kolejki priorytetowej. Element kolejki, który reprezentuje poprzednią wartość, od tego momentu staje się „śmieciem" w kolejce, dlatego przy usuwaniu elementów z kolejki sprawdzamy, czy nie natrafiamy na „śmieć", i ewentualnie pomijamy jego przetwarzanie (patrz instrukcja `break` w kodzie).

Jeden z błędów, jaki można tu popełnić, to użycie zwykłej kolejki w miejscu kolejki priorytetowej. Taka implementacja znajduje się w pliku `dijkstra_wolna.cpp`. Wprawdzie jest ona poprawna, jednak czasem działa znacznie wolniej.

Twoim zadaniem jest napisanie programu, który będzie generować testy, dla których wewnętrzna pętla **for** w drugim programie wykona *co najmniej* 20 razy więcej obrotów niż analogiczna pętla w pierwszym programie.

Wejście

W pierwszym i jedynym wierszu wejścia znajduje się słowo `sciezki`, po którym następuje liczba całkowita n ($300 \leq n \leq 100\,000$).

Wyjście

Twój program powinien wypisać wejście dla programów `dijkstra_poprawna.cpp` oraz `dijkstra_wolna.cpp` w następującym formacie. W pierwszym wierszu wyjścia powinny znaleźć się: liczba n wczytana z wejścia oraz liczba całkowita m ($1 \leq m \leq 5n$). Liczba n oznacza liczbę wierzchołków, zaś m — liczbę krawędzi grafu. W kolejnych m wierszach powinny znaleźć się opisy poszczególnych krawędzi grafu. Opis jednej krawędzi powinien składać się z trzech liczb całkowitych a_i, b_i, c_i ($1 \leq a_i, b_i \leq n$, $a_i \neq b_i$, $1 \leq c_i \leq 10^9$). Dla każdych dwóch wierzchołków może istnieć co najwyżej jedna krawędź, która je łączy. Każda krawędź grafu powinna zostać wypisana dokładnie raz. Musi istnieć co najmniej jedna krawędź o końcu w wierzchołku 1.