Zawody III stopnia

opracowania zadań

Piotr Chrząstowski — autor zadania, opracowanie
Marcin Sawicki — program wzorcowy

Lollobrygida

W fabryce poduszkowców do budowy torów testowych używa się standardowych bloków o różnych wysokościach, ustawianych jeden za drugim. W idealnie zbudowanym torze, zwanym **lollobrygidą**, *nigdy nie występują obok siebie dwa bloki jednakowej wysokości, nigdy też trzy kolejne bloki nie mają kolejno coraz większych, albo coraz mniejszych wysokości.*

Mówiąc bardziej formalnie, niech $h_1, \ldots, h_n$ oznacza ciąg wysokości kolejnych bloków należących do toru. Jeśli dla każdego $1 \leqslant i \leqslant n-2$ zachodzi:

- $h_i < h_{i+1}$ *i* $h_{i+1} > h_{i+2}$ *lub*
- $h_i > h_{i+1}$ *i* $h_{i+1} < h_{i+2}$,

to taki tor można nazwać **lollobrygidą**.

Przykład

Z zestawu 5 bloków o wysokościach 3, 3, 3, 5, 2 nie da się zbudować lollobrygidy, gdyż albo musiałyby stać w niej obok siebie dwa bloki wysokości 3, albo musi się w niej pojawić jedna z niedozwolonych sekwencji 2, 3, 5 lub 5, 3, 2.
A oto przykład lollobrygidy, poprawnie zbudowanej z innego zestawu bloków: 3, 2, 5, 2, 3, 1.
Z tego zestawu można też zbudować inne lollobrygidy.

Zadanie

Napisz program, który wczyta z pliku tekstowego `LOL.IN` *liczbę zestawów danych i dla każdego zestawu:*

- *wczyta liczbę bloków oraz wysokości poszczególnych bloków,*
- *stwierdzi, czy z podanego zestawu można zbudować lollobrygidę,*
- *zapisze wynik w pliku tekstowym* `LOL.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `LOL.IN` *znajduje się liczba całkowita d, $1 \leqslant d \leqslant 100$, równa liczbie zestawów danych. W następnym wierszu pliku* `LOL.IN` *zaczyna się pierwszy zestaw danych.*

W pierwszym wierszu każdego zestawu danych znajduje się liczba całkowita n, $3 \leqslant n \leqslant 1\,000\,000$. Jest to liczba bloków w tym zestawie.

W kolejnych n wierszach znajdują się wysokości bloków. Każdy z tych wierszy zawiera jedną liczbę całkowitą h równą wysokości odpowiedniego bloku, $1 \leqslant h \leqslant 10^9$.

Kolejne zestawy danych następują bezpośrednio po sobie.

Wyjście

Plik tekstowy `LOL.OUT` *powinien zawierać dokładnie d wierszy, po jednym dla każdego zestawu danych. W i-tym wierszu pliku* `LOL.OUT` *powinien być zapisany jeden wyraz:*

- *TAK, jeżeli z i-tego zestawu bloków można zbudować lollobrygidę,*
- *NIE, w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `LOL.IN`*:*
```
2
```

```
5
3
3
3
5
2
6
3
3
1
5
2
2
```
poprawną odpowiedzią jest plik tekstowy `LOL.OUT`*:*

```
NIE
TAK
```

ROZWIĄZANIE

Zadanie o lollobrygidzie w dużej części sprowadzało się do znanego i opisywanego w literaturze zadania o wyborze lidera. Zadanie to formułuje się następująco: Stwierdzić, czy w ciągu $a_1, \ldots, a_n$ istnieje wartość, która występuje w nim więcej niż $\frac{n}{2}$ razy. Jeżeli taka wartość występuje, to nazywamy ją liderem.

Okazuje się bowiem, że zachodzi następujące twierdzenie:

Twierdzenie. Z elementów $a_1, \ldots, a_n$ można ułożyć lollobrygidę wtedy i tylko wtedy, gdy zachodzi jeden z dwóch przypadków:

(1) W ciągu $a_1, \ldots, a_n$ nie występuje lider.

(2) Długość n jest nieparzysta i w ciągu $a_1, \ldots, a_n$ istnieje lider występujący w nim dokładnie $\frac{n+1}{2}$ razy, a pozostałe elementy ciągu $a_1, \ldots, a_n$ są albo wszystkie większe, albo wszystkie mniejsze od lidera.

Dowód. Najpierw pokażemy, że jeżeli istnieje lider nie spełniający warunku (2), to lollobrygidy ułożyć się nie da, a potem, że jeśli ciąg spełnia którykolwiek z warunków (1) lub (2), to lollobrygidę ułożyć się da.

Załóżmy więc, że istnieje lider nie spełniający warunku (2). Jeżeli n jest parzyste, to lider występuje co najmniej $\frac{n}{2}+1$ razy, zatem przy każdym ułożeniu wyrazów ciągu gdzieś dwaj liderzy muszą ze sobą sąsiadować. Jeśli bowiem podzielimy ciąg na dwuelementowe bloki (a jest ich dokładnie $\frac{n}{2}$), to z zasady szufladkowej Dirichleta wynika, że przynajmniej w jednym bloku znajdą się dwaj liderzy, więc podany ciąg nie będzie lollobrygidą.

Jeżeli n jest nieparzyste i lider występuje co najmniej $\frac{n+1}{2}+1$ razy, to dzieląc ciąg $a_1, \ldots, a_n$ na bloki dwuelementowe, otrzymamy takich bloków $\frac{n-1}{2}$ i zostanie jeszcze jeden blok jednoelementowy. Tak jak poprzednio, każda próba umieszczenia w takich blokach $\frac{n+1}{2}+1$ liderów musi się zakończyć tym, że dwaj liderzy znajdą się w jednym bloku obok siebie. Jeśli zaś n jest nieparzyste i lider występuje dokładnie $\frac{n+1}{2}$ razy, ale nie jest najmniejszą, ani największą wartością, to co prawda można wszystkich liderów rozmieścić w $\frac{n-1}{2}+1$ różnych blokach, ale żeby tor był lollobrygidą musimy umieścić ich wszystkich na nieparzystych pozycjach. Wtedy jednak na pewno wystąpi sytuacja, w której kolejne trzy wartości będą albo rosnące, albo malejące. Załóżmy bowiem dla symetrii, że $a_2 > a_1$ (a_1 jest wartością lidera). Wtedy $a_4 > a_3 = a_1$ (inaczej ciąg a_2, a_3, a_4 byłby malejący), $a_6 > a_5 = a_1$, itd. Okazałoby się więc, że a_1 jest najmniejszą wartością wbrew założeniu. Analogicznie przeprowadzamy rozumowanie, jeśli $a_2 < a_1$.

Pokazaliśmy więc pierwszą część twierdzenia. Pozostaje wykazać, że jeśli lidera nie ma lub jeśli jest lider, ale spełniający warunek (2), to lollobrygidę da się ułożyć z podanych wartości. Proste jest pokazanie, że jeśli lider spełnia warunek (2), to lollobrygidę daje się ułożyć. Wystarczy na wszystkich nieparzystych pozycjach ulokować lidera, a pozostałe elementy umieścić dowolnie na parzystych pozycjach. Warunek lollobrygidy w oczywisty sposób jest spełniony.

Zostało zatem do wykazania, że jeśli lidera nie ma, to lollobrygidę zawsze się da ułożyć. Dowód będzie indukcyjny ze względu na długość ciągu n.

Wzmocnimy najpierw nieco tezę — jest to częsty chwyt przy dowodach indukcyjnych; korzystamy przecież wtedy z mocniejszego założenia indukcyjnego. Pokażemy zatem, że jeśli lidera w ciągu $a_1, \ldots, a_n$ nie ma, to istnieje dla niego lollobrygida $b_1, \ldots, b_n$, i to taka, że $b_1 \neq b_n$.

Bazę indukcji pokażemy dla wartości $n = 2$. To nie szkodzi, że w sformułowaniu zadania $n \geqslant 3$. Jeżeli uda się nam zacząć indukcję od mniejszej wartości — tym lepiej. Twierdzenie staje się trochę ogólniejsze, a zazwyczaj dla mniejszych wartości n łatwiej jest udowodnić bazę indukcji. Faktycznie, załóżmy, że w ciągu a_1, a_2 nie ma lidera. Wobec tego $a_1 \neq a_2$, zatem ciąg a_1, a_2 jest lollobrygidą: nie ma w nim ani dwóch sąsiadujących, równych sobie elementów, jak również żadne trzy sąsiednie elementy nie są posortowane — po prostu nie istnieją.

Załóżmy teraz, że teza zachodzi dla każdego ciągu $a_1, \ldots, a_k$, $k < n$. Rozbijmy dowód na dwa przypadki: n nieparzyste i n parzyste.

Najpierw n **nieparzyste**. Niech m będzie modą ciągu $a_1, \ldots, a_n$. (Moda ciągu $a_1, \ldots, a_n$ jest to taka wartość, która w ciągu powtarza się najczęściej. Jeśli takich wartości jest więcej niż jedna, to każda z tych wartości jest modą. Lider oczywiście zawsze jest modą, ale nie na odwrót.) Usuńmy z ciągu $a_1, \ldots, a_n$ jeden element — modę. W powstałym $(n-1)$-elementowym ciągu nie ma lidera, bo przy nieparzystym n, co najwyżej $\frac{n-1}{2}$ elementów ciągu mogło być równych modzie nie będącej liderem i żadna inna wartość nie mogła wystąpić w nim więcej razy.

Taki ciąg po usunięciu jednego elementu będącego modą spełnia założenie indukcyjne: jest krótszy od n i nie zawiera lidera, zatem można z niego utworzyć lollobrygidę $b_1, \ldots, b_{n-1}$ taką, że $b_1 \neq b_{n-1}$. Załóżmy, że $b_1 < b_2$, a co za tym idzie, $b_{n-2} < b_{n-1}$ ($n-1$ jest parzyste, a kolejne wyrazy są na przemian od siebie większe i mniejsze). W przypadku odwrotnym, kiedy $b_1 > b_2$ i $b_{n-2} > b_{n-1}$, rozumowanie będzie analogiczne. Jeśli teraz $m > b_1$, to wstawiając m przed ciąg $b_1, \ldots, b_{n-1}$ otrzymujemy lollobrygidę, jeśli nie, to sprawdzamy, czy $m < b_{n-1}$. Jeśli tak, to lollobrygidę otrzymamy wstawiając m na koniec. Pozostał więc przypadek $b_1 \geqslant m \geqslant b_{n-1}$, przy czym jedna z tych nierówności musi być ostra. Wstawienie m z tej strony, z której nierówność jest ostra, między skrajny element, a jego sąsiada, spowoduje, że otrzymamy lollobrygidę o n elementach. To kończy dowód dla n nieparzystego.

Jeśli n jest **parzyste**, to mamy trochę inną sytuację, bo usunięcie mody może doprowadzić nas do sytuacji, w której pojawi się lider w ciągu $n-1$-elementowym i nie będzie można skorzystać z założenia indukcyjnego. Poradzimy sobie z tym przypadkiem zauważając, że tak może być jedynie wtedy, gdy w ciągu występują tylko dwie wartości i każda z nich jest modą. Wtedy nie musimy stosować indukcji — wystarczy ustawić jedne wartości na nieparzystych, a drugie na parzystych miejscach i mamy lollobrygidę. W każdym innym przypadku po usunięciu z ciągu, w którym nie było lidera, jednego elementu o wartości mody, lidera nadal nie będzie. Możemy wtedy stosować założenie indukcyjne. Załóżmy więc, że ciąg $b_1, \ldots, b_{n-1}$ jest lollobrygidą taką, że $b_1 \neq b_{n-1}$. Bez utraty ogólności możemy założyć, że $b_1 < b_2$, a co za tym idzie, $b_{n-2} > b_{n-1}$, gdyż $n-1$ jest nieparzyste. Przypadek przeciwny ma analogiczny dowód.

Jeżeli teraz $m > b_1$ lub $m > b_{n-1}$, to wstawiając m na początek lub odpowiednio na koniec, uzyskujemy lollobrygidę. Pozostaje zatem do rozważenia przypadek, gdy m nie przekracza żadnego ze skrajnych elementów. Ponieważ są one różne, więc m od jednego z nich musi być ostro mniejsze. Dajmy na to, że $m < b_1$. Wtedy wstawiając m między b_1 a b_2, otrzymujemy lollobrygidę. Analogicznie postępujemy, gdy $m < b_{n-1}$.

Wyczerpaliśmy tym samym wszystkie przypadki, za każdym razem pokazując, że lollobrygida jest możliwa do ułożenia. Twierdzenie zostało tym samym udowodnione.

Uff! Mamy już w ręku ważny wynik, który ułatwi rozwiązanie zadania w efektywny sposób. Pozostaje więc sprawdzić, czy lider istnieje i jeżeli nie, to odpowiedź jest TAK, a jeśli lider istnieje, to wystarczy upewnić się, czy nie spełnia warunku (2). Jeżeli spełnia, to odpowiedź jest TAK, a jeśli nie spełnia, to odpowiedź jest NIE.

Omówmy tu 4 różne algorytmy stwierdzania istnienia lidera.

Pierwszy algorytm, być może najprostszy do wymyślenia, polega na posortowaniu danych i znalezieniu w posortowanym ciągu najdłuższej sekwencji tych samych elementów. Koszt dobrego sortowania jest proporcjonalny do $n \log n$, koszt znalezienia najdłuższej stałej sekwencji jest proporcjonalny do n. Łącznie mamy więc algorytm o złożoności rzędu $n \log n$. Nieźle, ale nie optymalnie.

Drugi algorytm bazuje na spostrzeżeniu, że jeśli w ciągu jest lider, to jest on równy medianie ciągu, czyli elementowi, który w posortowanym ciągu wystąpiłby pośrodku (dla n parzystego jako medianę definiuje się dowolny z dwóch elementów środkowych). Algorytm zatem rozbija się na dwie części:

wyznaczenie mediany i sprawdzenie, czy mediana jest liderem. Ta druga część jest bardzo prosta. Wyznaczenie mediany jest bardziej kłopotliwe, o ile nie chcemy korzystać z sortowania.

Medianę można wyznaczyć w czasie pesymistycznym proporcjonalnym do n, ale nie jest to proste. Zadanie to realizuje na przykład algorytm Bluma, Floyda, Pratta, Rivesta, Tarjana, zwany algorytmem piątek lub algorytmem pięciu — każda z nazw jest uzasadniona. Algorytm ten opisany jest np. w książce [10]. Problem polega na tym, że koszt tego algorytmu, choć liniowy, obarczony jest dużą stałą i dla potrzeb naszego zadania algorytm ten raczej się nie nadaje. Przy podanych ograniczeniach na rozmiar danych uzyskanie za pomocą tego algorytmu lepszego czasowo rozwiązania, niż przez sortowanie, wydaje się trudne, a być może jest wręcz niemożliwe. Pamiętajmy bowiem, że choć teoretycznie złożoność jednego algorytmu może być lepsza od złożoności drugiego, to wcale nie znaczy, że zawsze należy stosować ten pierwszy. Na przykład dla krótkich ciągów danych (rzędu kilku-kilkunastu) nie należy sortować quicksortem, tylko którymś z algorytmów o złożoności kwadratowej, choćby przez proste wstawianie.

Z praktycznego punktu widzenia lepiej byłoby tu zastosować opisany również w [10] algorytm Hoare'a wyznaczania mediany. Jego idea jest podobna do pomysłu sortowania quicksortem. Losuje się bowiem jedną z wartości ciągu, a następnie rozrzuca pozostałe wartości na 3 grupy: elementów mniejszych, równych i większych od tej wartości, po czym szuka się elementu o odpowiednim numerze w jednej z tych grup, tą samą zresztą metodą. Ten algorytm ma kwadratową złożoność pesymistyczną (gdy zawsze będziemy mieli pecha i wybierali element skrajny jako wartość, względem której rozrzucamy), ale średnio działa w czasie liniowym. W praktyce doskonale nadaje się do znajdowania mediany i jest dość prosty do zakodowania — własność bardzo ważna w trakcie zawodów.

Trzeci algorytm, trochę niepewny, bo nie dający dobrych rezultatów ze stuprocentową pewnością, to algorytm probabilistyczny polegający po prostu na kilkukrotnym wylosowaniu dowolnej wartości i sprawdzeniu, czy jest ona liderem. Zauważmy, że jeśli lider istnieje, to z prawdopodobieństwem większym niż $\frac{1}{2}$ wylosujemy właśnie jego. Jeżeli na przykład po dziesięciu losowaniach nie natrafimy na lidera, to uznajemy, że lidera nie ma i dajemy odpowiedź TAK. Prawdopodobieństwo błędu jest mniejsze niż $\frac{1}{1000}$. W praktyce tego typu rozwiązania są stosowane i na olimpiadzie czasami bardzo skuteczne. Tutaj wadą jest konieczność wielokrotnego przeglądania całego ciągu wejściowego. Im większą chcemy mieć pewność, tym więcej to kosztuje. Testy były tak przygotowane, żeby z dużym prawdopodobieństwem spowodować podanie złej odpowiedzi, albo przekroczenie limitu czasowego, co najmniej w jednym z zestawów. Ale podobnie jak zawodnicy, którzy użyli tego algorytmu do rozwiązania naszego zadania, nie mogli być pewni, czy otrzymają dobre wyniki, tak i organizatorzy, sprawdzając zadanie, nie mogli być pewni, czy wychwycą niepewne rozwiązanie.

Czwarty algorytm, wzorcowy, i bardzo elegancki, którego poprawność jest wysoce nieoczywista. Algorytm ten opisany jest w książce [23]. Ze względu na prostotę algorytmu podamy tutaj jego pełny kod pascalowy.

```
{ Zakładamy, że elementy ciągu podane są w tablicy A[1..n].
  Algorytm stwierdza, czy w tablicy A znajduje się lider. }
ile:=1; l:=A[1];  { l - kandydat na lidera }
for i:=2 to n do
 if A[i]=l then ile:=ile+1
 else if ile = 1  then l:=A[i]
                  else ile:=ile-1;
{ Po tej pętli, jeśli w A jest lider, to jest on równy l.
  Sprawdzamy więc, czy l jest liderem. }
ile:=0;
for i:=1 to n do if A[i]=l then ile:=ile+1;
if ile > n div 2 then { lider jest }
                 else { lidera nie ma }
```

Uwaga: aby użyć tego algorytmu do rozwiązania naszego zadania, w drugiej pętli należy dodać fragment sprawdzający warunek (2) z udowodnionego twierdzenia. Fragment ten pominęliśmy, aby nie popsuć przejrzystości tekstu.

Nie od razu widać, dlaczego ten algorytm miałby dawać poprawne odpowiedzi. Najlepiej się o tym przekonać próbując skonstruować kontrprzykład. Po kilku próbach okaże się to niemożliwe. Pełny dowód poprawności algorytmu jednak nie jest banalny. Przy okazji warto zauważyć, że o ile poprzednie algorytmy potrzebowały tablicy do zapamiętania wszystkich wartości ciągu, to tutaj możemy z tablicy zrezygnować i wczytywać na bieżąco wartości z pliku. Dwukrotne wczytanie wystarcza, żeby stwierdzić istnienie lidera. Algorytm ten ma zatem koszt czasowy liniowy, a pamięciowy stały.

Pozostaje jeszcze wyjaśnić, dlaczego tor o podanych własnościach nazwaliśmy lollobrygidą. Narciarze zapewne wiedzą, że taką nazwę nosi jedna z bardzo muldziastych tras narciarskich w Szczyrku. Dlaczego jednak trasa ta została nazwana nazwiskiem słynnej włoskiej aktorki, pozostawmy dociekliwości czytelników.

TESTY

Testy i czasy odcięcia zostały tak dobrane, aby możliwie jak najdokładniej oddzielić algorytmy liniowe od algorytmów o czasie działania rzędu $n \log n$ i wyeliminować programy błędne (np. zgadujące odpowiedź).

Każdy test składa się z 30 przypadków. Pierwsze 4 testy, to testy poprawnościowe, zawierające same proste przypadki ($n \leqslant 22$). Następne 6 testów, to testy wydajnościowe. Zawierają po kilka przypadków dużych danych ($n = 99999\frac{1}{2} \pm \frac{1}{2}$), pozostałe przypadki są podobne co do wielkości do pierwszych czterech testów.

Oto opisy poszczególnych testów:

- LOL1.IN — małe dane, n parzyste.
- LOL2.IN — małe dane, n parzyste, jeśli odpowiedzią ma być „TAK”, to żadne $\frac{n}{2}$ elementów się nie powtarza (tzn. nie ma „prawie lidera”), wówczas konstrukcja lollobrygidy może być dla niektórych algorytmów nieco trudniejsza.
- LOL3.IN — małe dane, n nieparzyste, nie ma lidera, o ile odpowiedzią ma być „TAK”.
- LOL4.IN — małe dane, n nieparzyste, w każdym przypadku istnieje lider, czyli jeśli odpowiedzią jest „TAK”, to wszystkie inne elementy są od niego albo mniejsze, albo większe.

W testach wydajnościowych występują przemieszane wszystkie 4 powyższe przypadki parzystości i istnienia, lub nie, lidera.

- LOL5.IN — 3 przypadki duże, 27 małych, dużo różnych wartości, kolejność elementów losowa.
- LOL6.IN — 3 przypadki duże, 27 małych, dużo różnych wartości, kolejność elementów rosnąca. Chodzi o wyeliminowanie pewnych nieefektywnych implementacji algorytmu Hoare'a lub Quicksortu, które w takim przypadku zachowują się kwadratowo.
- LOL7.IN — 3 przypadki duże, 27 małych, dużo różnych wartości, kolejność elementów malejąca.
- LOL8.IN — 3 przypadki duże, 27 małych, kolejność rosnąca, zarówno przypadki z dużą ilością różnych wartości, jak i z kilkunastoma wartościami powtarzającymi się wielokrotnie oraz z zaledwie trzema różnymi wartościami.
- LOL9.IN — jak wyżej, kolejność malejąca.
- LOL10.IN — 8 dużych przypadków, 22 małe, każdy z przypadków składa się jedynie z trzech różnych wartości, albo przemieszanych, albo posortowanych rosnąco lub malejąco.

Czasy odcięcia były dobrane tak, aby więcej niż dwukrotnie przewyższały czas działania wzorcowego rozwiązania, ale dla dużych testów powodowały odcięcie przy korzystaniu z procedur sortujących. Małe testy doczepione zostały do dużych aby wyeliminować algorytmy zgadujące odpowiedzi, bowiem jedynie rozwiązanie kompletu 30 przypadków zaliczało test.

Marcin Mucha
autor zadania, opracowanie

Paweł Wolff
program wzorcowy

Jajka

Wiadomo, że jajko zrzucone z wystarczająco dużej wysokości brzydko się rozbija. Dawniej wystarczyła wysokość jednego piętra, ale genetycznie podrasowane kury znoszą jajka nie tłukące się nawet po zrzuceniu z wysokości 100 000 000 pięter. Badania nad wytrzymałością jajek prowadzi się wykorzystując drapacze chmur. Opracowano specjalną skalę wytrzymałości jajek: jajko ma wytrzymałość k pięter, jeśli zrzucone z k-tego piętra nie rozbija się, ale zrzucone z $(k+1)$-ego już tak. W przypadku gdy drapacz chmur, którym dysponujemy, ma n pięter, przyjmujemy, że jajko rozbija się, gdy zrzucimy je z $(n+1)$-ego piętra. Przyjmujemy też, że każde jajko zrzucone z piętra o numerze 0 nie rozbija się.

Kierownik laboratorium postanowił wprowadzić oszczędności w procesie badawczym. Ograniczył on liczbę jajek, które wolno rozbić w trakcie eksperymentu mającego na celu ustalenie wytrzymałości jajek danego gatunku. Dodatkowo należy zminimalizować liczbę zrzutów jajek. Oznacza to, że mając do dyspozycji pewną liczbę jajek danego gatunku i drapacz chmur należy, w jak najmniejszej liczbie prób stwierdzić, jaka jest wytrzymałość jajek danego gatunku.

Zadanie

Twoim zadaniem jest napisanie modułu zawierającego trzy procedury (funkcje w przypadku języka C/C++):

- `nowy_eksperyment` — *ta procedura będzie wywoływana na początku każdego nowego eksperymentu (może być ich więcej niż jeden);*
- `daj_pytanie` — *ta procedura służy do zadawania pytania, czy jajko wytrzyma zrzucenie z określonego piętra, czy też rozbije się; w tym celu należy w opisanej dalej zmiennej globalnej* `pietro` *umieścić numer piętra, z którego ma być zrzucone jajko,*
- `analizuj_odpowiedz` — *ta procedura powinna odczytać wartość zmiennej globalnej* `odpowiedz`*, która zawiera odpowiedź na ostatnio zadane pytanie (opis tej zmiennej znajduje się w kolejnym paragrafie) i dokonać analizy tej odpowiedzi — jeśli w wyniku tej analizy zostanie określona wytrzymałość jajka, to procedura ta powinna ten fakt zasygnalizować poprzez nadanie odpowiednich wartości zmiennym globalnym* `x` *oraz* `wiem` *(szczegóły znajdują się w kolejnym paragrafie).*

Program nadzorujący przebieg eksperymentów wywoła napisaną przez Ciebie procedurę `nowy_eksperyment` *na początku każdego nowego eksperymentu, po czym cyklicznie będzie wykonywał następujące czynności:*

- *wywołanie procedury* `daj_pytanie`*,*
- *odpowiadanie na pytanie,*
- *wywołanie procedury* `analizuj_odpowiedz`*,*

aż do momentu, gdy Twój program stwierdzi, że zna wytrzymałość jajek gatunku używanego w danym eksperymencie (tzn. procedura `analizuj_odpowiedz` *umieści odpowiednią wartość w zmiennej globalnej* `wiem`*).*

Uwaga: *Nie zakładaj, że program nadzorujący przebieg eksperymentów faktycznie ustala pewną wytrzymałość jajka przed rozpoczęciem danego eksperymentu. Może on dobierać ją w trakcie trwania eksperymentu w taki sposób, aby pasowała do wszystkich wcześniej udzielonych odpowiedzi oraz aby zmusić Twój program do zadania jak największej liczby pytań. Tak więc, powinieneś dążyć do tego, aby liczba pytań, jakie Twój program będzie musiał zadać w najgorszym przypadku, była jak najmniejsza.*

Komunikacja

Komunikacja pomiędzy napisanym przez Ciebie modułem, a programem nadzorującym przebieg eksperymentów, odbywa się poprzez zmienne globalne.

Liczba pięter drapacza zapisana będzie w zmiennej globalnej `wysokosc` *typu* `Longint` *(w przypadku języka C/C++ jest to typ* `long int`*). Będzie to dodatnia liczba całkowita, nie większa niż 100 000 000.*

Maksymalna liczba jajek, które można stłuc w trakcie trwania eksperymentu zapisana będzie w zmiennej globalnej `jajka` *typu* `Integer` *(w przypadku języka C/C++ jest to typ* `int`*). Będzie to dodatnia liczba całkowita, nie większa niż 1000.*

Zadanie pytania, czy jajko wytrzyma zrzucenie z k-tego piętra, w procedurze `daj_pytanie` *polega na przypisaniu zmiennej globalnej* `pietro` *typu* `Longint` *(w przypadku języka C/C++ jest to typ* `long int`*) liczby k.*

Odpowiedź na zadane pytanie jest umieszczana w zmiennej globalnej `odpowiedz` *typu* `Boolean` *(w przypadku języka C/C++ jest to typ* `int`*). Twierdzącej odpowiedzi na zadane pytanie (tzn. jajko wytrzyma) odpowiada wartość* `TAK`*, a przeczącej (tzn. jajko rozbije się) odpowiada* `NIE`*, gdzie* `TAK` *i* `NIE` *są stałymi o wartościach, odpowiednio,* `true` *i* `false` *(w przypadku języka C/C++ są to makra o wartościach, odpowiednio,* `1` *i* `0`*).*

W przypadku znalezienia wytrzymałości jajka Twój program powinien w procedurze `analizuj_odpowiedz` *zapisać do zmiennej globalnej* `wiem` *typu* `Boolean` *(w przypadku języka C/C++ jest to typ* `int`*)* `TAK` *oraz w zmiennej globalnej* `x` *typu* `Longint` *(w przypadku języka C/C++ jest to typ* `long int`*) znalezioną wytrzymałość jajka.*

Katalogi i pliki

Programujący w Pascalu powinni przygotowywać swoje rozwiązanie w katalogu `JAJPAS`*, natomiast programujący w C i C++ w katalogu* `JAJC`*.*

W katalogu `JAJPAS` *(odpowiednio* `JAJC`*) znajdziesz następujące pliki:*

- `JAJmod.pas` *(odpowiednio* `JAJmod.h` *i* `JAJmod.c`*) — moduł zawierający definicje stałych* `TAK` *i* `NIE` *oraz deklaracje zmiennych globalnych,*
- `JAJ.pas` *— szkielet modułu znajdującego wytrzymałość jajka; powinieneś uzupełnić ten plik definicjami funkcji* `nowy_eksperyment`*,* `daj_pytanie` *i* `analizuj_odpowiedz` *(dla piszących w C lub C++ nagłówki powyższych funkcji znajdują się w pliku* `JAJ.h`*, musisz napisać plik* `JAJ.c` *lub* `JAJ.cpp` *z definicjami tych funkcji),*
- `test.pas` *(odpowiednio* `test.c`*) — przykładowy program nadzorujący przebieg eksperymentów i korzystający z Twojego modułu.*

Wyjście

Wynikiem Twojej pracy powinien być zapisany na dyskietce tylko jeden plik: `JAJ.pas`*,* `JAJ.c` *lub* `JAJ.cpp`*.*

ROZWIĄZANIE

Na początku eksperymentu wiemy, że wytrzymałość jajek danego gatunku mieści się w przedziale $[0, \texttt{wysokosc}]$. Jeśli na pewnym etapie eksperymentu wiemy, że wytrzymałość mieści się w przedziale $[a, b]$, to czego dowiemy się po zrzuceniu jajka z piętra o numerze p? Oczywiście będziemy wybierać $a < p \leqslant b$, gdyż dla innych wartości p jest jasne, czy jajko się rozbije, czy nie. W przypadku, gdy jajko się rozbije będziemy wiedzieli, że jego wytrzymałość zawiera się w przedziale $[a, p-1]$. Jeżeli natomiast jajko wytrzyma próbę, oznacza to, że jego wytrzymałość mieści się w przedziale $[p, b]$. Oczywiście eksperyment możemy zakończyć tylko wtedy, gdy długość przedziału, w jakim mieści się wytrzymałość jajek danego gatunku, wynosi 0. Zastanówmy się teraz, jak duży przedział wytrzymałości możemy sprawdzić mając do dyspozycji k jajek i n prób (tę maksymalną długość przedziału oznaczmy przez $T(n, k)$). Oczywiście mamy:

$$T(n, 0) = T(0, k) = 0, \text{ dla } n, k \geqslant 0,$$

bo nie mając żadnych jajek lub prób, niewiele możemy zdziałać.

Jeżeli chcemy znaleźć wytrzymałość jajek pewnego gatunku w n próbach mając do dyspozycji k jajek i wiemy, że ich wytrzymałość mieści się w przedziale $[a, b]$, to musimy zrzucić jajko z piętra o numerze nie większym niż $a + T(n-1, k-1) + 1$ (w przeciwnym razie, w przypadku rozbicia jajka, pozostaje nam $n-1$ prób i $k-1$ jajek, lecz wytrzymałość mieści się w przedziale, którego długość jest większa niż $T(n-1, k-1)$) oraz nie mniejszym niż $b - T(n-1, k)$ (w przeciwnym razie, w przypadku, gdy jajko wytrzyma próbę, pozostaje nam $n-1$ prób i k jajek, lecz wytrzymałość mieści się w przedziale o długości większej niż $T(n-1, k)$). Aby oba te warunki mogły być spełnione, długość przedziału $[a, b]$ nie może być większa od $T(n-1,k-1)+1+T(n-1,k)$. W takim przypadku, jajko można zrzucić z piętra o numerze $min(a+T(n-1,k-1)+1,b)$, a zatem:

$$T(n,k)=T(n-1,k)+T(n-1,k-1)+1,\ \text{dla } n,k\geqslant 1.$$

Ta formuła bardzo przypomina poniższą:

$$\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1}$$

To sugeruje nam zbadanie związku liczb $T(n,k)$ ze współczynnikami dwumianowymi $\binom{n}{k}$. Łatwo sprawdzić (troszkę trudniej zgadnąć), że:

$$T(n,k)=\sum_{i=1}^{k}\binom{n}{i},\ \text{dla } n,k\geqslant 0.$$

Porównując zapisy różnych $T(n,k)$ łatwo otrzymujemy:

(1) $T(n+1,k)=2T(n,k)+1-\binom{n}{k}$, dla $n,k\geqslant 0$;

(2) $T(n-1,k)=\frac{1}{2}(T(n,k)+\binom{n-1}{k}-1)$, dla $n\geqslant 1, k\geqslant 0$;

(3) $T(n,k-1)=T(n,k)-\binom{n}{k}$, dla $n\geqslant 0, k\geqslant 1$.

Przy pomocy wzoru (1) możemy wyznaczyć najmniejsze n takie, że $T(n,\texttt{jajka})\geqslant \texttt{wysokosc}$. Jest to, jak łatwo zauważyć, minimalna liczba prób jaka wystarcza do wyznaczenia wytrzymałości jajek w danym eksperymencie. Czas wyznaczenia liczby n jest oczywiście liniowy ze względu na tę liczbę. Przy pomocy wzorów (2), (3) możemy z kolei wyznaczyć w czasie stałym numer piętra, z którego należy zrzucić jajko (szczegóły można znaleźć w treści programu).

Jak liczyć symbole Newtona

Rozwiązanie wzorcowe wydaje się proste do zaprogramowania. Jest w nim jednak pewien haczyk. Na przykład, w jaki sposób obliczać $\binom{n}{k}$ znając $\binom{n-1}{k}$? Nie możemy po prostu pomnożyć przez n i podzielić przez $n-k$, bo przekroczymy zakres. Można jednak poradzić sobie z tym problemem stosując częściowe dzielenie. Wiemy, że wynik naszych obliczeń będzie liczbą całkowitą. Można więc w ułamku $\frac{n\binom{n-1}{k}}{n-k}$ skrócić najpierw lewą część ($\frac{n}{n-k}$), a potem prawą (szczegóły można znaleźć w treści programu).

Inne rozwiązania

Czy można uniknąć zgadywania tajemniczego wzoru na $T(n,k)$? Chyba tak. Możemy przecież wyliczać $T(n,k)$ z formuły rekurencyjnej:

$$T(n,k)=T(n-1,k)+T(n-1,k-1)+1,\ \text{dla } n,k\geqslant 1,$$

tzn. obliczać $T(n,0),T(n,1),\dots,T(n,\texttt{jajka})$ dla kolejnych n, aż $T(n,\texttt{jajka})\geqslant\texttt{wysokosc}$.

Następnie postępować tak, jak w rozwiązaniu wzorcowym, przy czym jeśli mamy policzone wartości

$$T(n,0),\dots,T(n,\texttt{jajka}),$$

to

$$T(n-1,0),\dots,T(n-1,\texttt{jajka})$$

możemy uzyskać z wzoru:

$$T(n-1,k) = T(n,k) - T(n-1,k-1) - 1, \text{ dla } k > 0 \text{ i } T(n-1,0) = 0$$

(oczywiście można pamiętać wszystkie wyliczone wartości i wtedy korzystanie z powyższego wzoru nie jest konieczne).

Jaka jest złożoność takiego algorytmu? Oczywiście $O(\texttt{jajka}\cdot n)$, gdzie n jest najmniejszą taką liczbą, że $T(n, \texttt{jajka}) \geqslant \texttt{wysokosc}$. Pozostaje pytanie: jak duże może być n? Można znaleźć dość dobre oszacowanie korzystając z mądrych wzorów, wystarczy jednak prosty rachunek, aby się przekonać, że iloczyn $n \cdot \texttt{jajka}$ nigdy nie jest zbyt duży.

Jeśli $\texttt{jajka} = 1$, to oczywiście $n = \texttt{wysokosc}$. Ten przypadek można jednak rozpatrywać osobno. Wystarczy rzucać jajko z kolejnych pięter.

Jeśli $\texttt{jajka} = 2$, to mamy:

$$T(n,2) = \binom{n}{1} + \binom{n}{2} = n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2} > \frac{n^2}{2},$$

a więc wystarczy $n = \sqrt{2wysokosc} \leqslant 10000\sqrt{2} < 15000$, co daje łącznie $\texttt{jajka} \cdot n = 30000$ operacji.

Dla $\texttt{jajka} \geqslant 3$ mamy:

$$T(n,k) \geqslant \binom{n}{1} + \binom{n}{2} + \binom{n}{3} = n + \frac{n(n-1)}{2} + \frac{n(n-1)(n-2)}{6} =$$

$$= n + \frac{n(n-1)(n+1)}{6} = \frac{n(n^2+5)}{6} > \frac{n^3}{6},$$

a więc wystarczy $n = \sqrt[3]{6wysokosc} \leqslant \sqrt[3]{600000000} \leqslant 900$ i łącznie najwyżej $900 \cdot \texttt{jajka}$ operacji. To już jest dobra złożoność, ale oczywiście można przeprowadzić podobny prosty rachunek dla $\texttt{jajka} \geqslant 4$ i otrzymać jeszcze lepsze oszacowanie.

Opis działania programu arbitra

Arbiter oszukuje

Arbiter wcale nie ustala na początku eksperymentu wytrzymałości jajka. Wczytuje natomiast z pliku testowego przedział $[min..max]$ wytrzymałości dopuszczalnych. Oznacza to z grubsza tyle, że jego odpowiedzi mają się zgadzać z pewną wytrzymałością z tego przedziału. Przykładowo, jeśli $min = max$, to arbiter faktycznie ustala na początku eksperymentu wytrzymałość jajka

Arbiter jest złośliwy

Arbiter nie ma wyboru przy udzielaniu odpowiedzi na pytanie, czy jajko zrzucone z k-tego piętra rozbije się, gdy $k \leqslant min$ lub $k > max$. W przeciwnym razie, jeśli któraś z odpowiedzi wymusi na module zawodnika zadanie większej liczby pytań niż jest to konieczne, to taka odpowiedź jest udzielana. Jeśli zaś przy obu odpowiedziach moduł zawodnika zachowuje szansę na wyznaczenie wytrzymałości jajek w odpowiedniej liczbie prób, to udzielana jest odpowiedź wymuszająca na module zawodnika zadanie co najmniej minimalnej wystarczającej liczby pytań. Jeśli obie odpowiedzi wymuszają zadanie co najmniej minimalnej wystarczającej liczby pytań, to udzielana jest odpowiedź losowa. Jeśli żadna odpowiedź nie ma tej właściwości (może się tak zdażyć wtedy, gdy liczby min i max różnią się mało, a zawodnik "wstrzeli" się w przedział $[min, max]$), to udzielana jest odpowiedź `TAK`.

Oczywiście jeśli arbiter odpowiedział, że jajko zbije się po zrzuceniu z piętra $k < max$, to od tego momentu $max = k - 1$. Analogicznie arbiter musi zwiększyć min, jeśli odpowiedział, że jajko się nie bije i $k > min$.

Arbiter wie wszystko

No dobrze, ale w jaki sposób można stwierdzić, czy po udzieleniu konkretnej odpowiedzi zawodnik będzie miał szansę na zmieszczenie się w minimalnej wystarczającej liczbie pytań? Można to zrobić, jeśli zna się wartości $T(n,k)$. Arbiter wylicza te wartości w taki sam sposób, jak moduł wzorcowy.

Diabelska strategia arbitra

Jak już zaznaczyliśmy na wstępie, przypadek $min = max$ jest wyjątkowo łagodny dla zawodnika. Na drugim biegunie leży przypadek $min = 0$, $max = wysokosc$. W tej sytuacji arbiter nie jest w żaden sposób ograniczony i gładko wymusza na module zawodnika, aby ten zadał możliwie najwięcej pytań. Ten przypadek będziemy nazywać diabelską strategią arbitra.

Czy arbiter gra w kości?

W niektórych przypadkach arbiter udziela odpowiedzi losowych. Czy to oznacza, że ten sam deterministyczny moduł uruchomiony dwa razy może uzyskać dwie różne liczby punktów? To zdecydowanie nie byłoby dobrze. Dlatego arbiter z pliku testowego oprócz wartości min i max wczytuje ciąg 100 losowych bitów, z którego korzysta przy udzielaniu odpowiedzi.

TESTY

Do sprawdzania rozwiązań zawodników użyto 11 testów, których opisy znajdują się poniżej.

- JAJ1.IN — małe wysokości drapacza, mało jajek, uczciwy arbiter,
- JAJ2.IN — średnie wysokości drapacza, różna liczba jajek, uczciwy arbiter,
- JAJ3.IN — małe wysokości drapacza, mało jajek, średniej trudności strategia udzielania odpowiedzi przez arbitra,
- JAJ4.IN — średnie wysokości drapacza, różna liczba jajek, średniej trudności strategia udzielania odpowiedzi przez arbitra,
- JAJ5.IN — duże wysokości drapacza, różna liczba jajek, średniej trudności strategia udzielania odpowiedzi przez arbitra,
- JAJ6.IN — małe wysokości drapacza, mało jajek, strategia diabelska,
- JAJ7.IN — średnie wysokości drapacza, różna liczba jajek, strategia diabelska,
- JAJ8.IN — duże wysokości drapacza, różna liczba jajek, strategia diabelska,
- JAJ9.IN — duże wysokości drapacza, różna liczba jajek, strategia diabelska,
- JAJ10.IN — duże wysokości drapacza, różna liczba jajek, strategia diabelska.

Grzegorz Jakacki
autor zadania

Marcin Mucha
opracowanie, program wzorcowy

Po-łamana

W prostokątnym układzie współrzędnych każdy punkt o współrzędnych całkowitych nazywamy **p-punktem***.*

Dowolny odcinek równoległy do jednej z osi współrzędnych, o różnych końcach będących **p-punktami***, nazywamy* **p-odcinkiem***. Rozważamy odcinki domknięte, tzn. końce należą do odcinka. Łamaną zbudowaną z k* **p-odcinków***, z których każde kolejne dwa są prostopadłe, nazywamy* **po-łamaną** *stopnia k.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `POL.IN` *opisy pewnej liczby p-odcinków oraz współrzędne dwóch różnych p-punktów A i B,*
- *wyznaczy minimalny stopień po-łamanej łączącej te dwa punkty i nie przecinającej żadnego z danych p-odcinków lub stwierdzi, że taka po-łamana nie istnieje,*
- *zapisze wynik do pliku tekstoweg* `POL.OUT`*.*

Wejście

Opis p-punktu składa się z dwóch nieujemnych liczb całkowitych oddzielonych pojedynczym odstępem, będących odpowiednio współrzędnymi x i y tego p-punktu. Liczby te należą do przedziału [0..1 000 000 000]. W pierwszym wierszu pliku wejściowego `POL.IN` *znajduje się tylko opis p-punktu A. W drugim wierszu znajduje się tylko opis p-punktu B. W trzecim wierszu zapisana jest dokładnie jedna nieujemna liczba całkowita n będąca liczbą p-odcinków, $1 \leqslant n \leqslant 50$. W każdym z kolejnych n wierszy znajdują się opisy dokładnie dwóch p-punktów, oddzielone pojedynczym odstępem. Są to współrzędne końców jednego p-odcinka.*

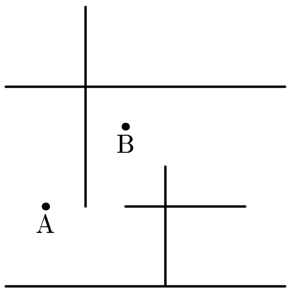
Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `POL.OUT` *powinna znaleźć się jedna liczba będąca minimalnym stopniem po-łamanej łączącej punkty A i B oraz nie przecinającej żadnego z zadanych p-odcinków, lub słowo* `BRAK`*, jeśli po-łamana o powyższych własnościach nie istnieje.*

Przykład

Dla pliku wejściowego `POL.IN`*:*

```
1 2
3 4
5
0 0 7 0
0 5 7 5
2 2 2 7
4 0 4 3
3 2 6 2
```



poprawną odpowiedzią jest plik wyjściowy `POL.OUT`*:*

```
5
```

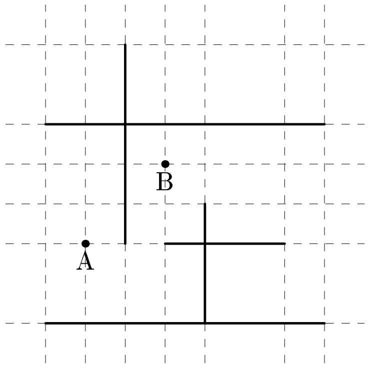
ROZWIĄZANIE WZORCOWE

Niech $x_1 < x_2 < \ldots$ będzie rosnąco uporządkowanym ciągiem (bez powtórzeń) współrzędnych x wszystkich końców odcinków z dodatkowo dodaną współrzędną x punktu B. Niech $y_1 < y_2 < \ldots$ będzie analogicznym ciągiem dla współrzędnej y. Rozważmy podział osi $X = (-\infty, \infty) = \bigcup_i X_i$, gdzie

$$X_1 = (-\infty, x_1), X_2 = \{x_1\}, X_3 = (x_1, x_2), X_4 = \{x_2\} \ldots$$

Analogicznie definiujemy podział osi Y. Podziały osi wyznaczają podział płaszczyzny na prostokąty otwarte, odcinki otwarte (być może nieograniczone) i punkty.

Rys. 1 Podział płaszczyzny dla przykładowego zestawu p-odcinków



Niech P_1 i P_2 będą dwoma elementami powyższego podziału. Jeśli z pewnego punktu $p \in P_1$ istnieje łamana o ustalonym stopniu i końcu w P_2, to z każdego innego punktu P_1 także. W szczególności za P_2 można przyjąć punkt B.

Oznacza to, że punkty w jednym elemencie podziału zachowują się identycznie, jeśli chodzi o istnienie łamanych o końcu w punkcie B. Rozwiązanie w zasadzie się narzuca. Trzeba skonstruować graf, w którym wierzchołkami są elementy podziału (ma on strukturę kraty, co ułatwia implementację), krawędzie zaś oznaczają możliwość przejścia z jednego elementu do innego. Następnie w tym grafie używamy procedury przeszukiwania wszerz (BFS), aby znaleźć łamaną minimalnego stopnia łączącą element zawierający punkt A z punktem B. Musimy jednak w procedurze BFS dokonać dwu modyfikacji:

(1) oddzielnie rozpatrywać wejścia pionowe i poziome, ponieważ ze względu na dalszą trasę nie są to sytuacje jednakowe,

(2) po dojściu do wierzchołka "pionowo" przechodzimy nie do jego najbliższych sąsiadów, ale do wszystkich wierzchołków, do których można z niego dojść poziomo (przypadek dojścia poziomego jest analogiczny) — w ten sposób algorytm ma złożoność przeszukiwania BFS, a pierwsze dojście do punktu B od razu daje odpowiedź na postawione w zadaniu pytanie.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 12 testów:

- POL0.IN — test z treści zadania,
- POL1.IN — spirala o 10 krawędziach,
- POL2.IN — spirala o 20 krawędziach,
- POL3.IN — spirala o 30 krawędziach,
- POL4.IN — spirala o 40 krawędziach,
- POL5.IN — spirala o 45 krawędziach,

- POL6.IN — spirala o 50 krawędziach,
- POL7.IN — test poprawnościowy,
- POL8.IN — test wydajnościowy o 20 krawędziach,
- POL9.IN — test wydajnościowy o 30 krawędziach,
- POL10.IN — test wydajnościowy o 40 krawędziach,
- POL11.IN — test wydajnościowy o 50 krawędziach.

Tomasz Waleń
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Agenci

W związku z ostatnimi wpadkami swoich agentów, Urząd Ochrony Bajtocji postanowił usprawnić działalność.

Największym dotychczasowym problemem było bezpieczne urządzanie spotkań agentów. Twój program ma pomóc w rozwiązaniu tego problemu. Dla podanego opisu sieci dróg Bajtocji oraz początkowej pozycji dwóch agentów, powinien stwierdzać, czy możliwe jest bezpieczne spotkanie tych agentów.

Żeby spotkanie uznać za bezpieczne agenci muszą przestrzegać następujących reguł:

- *agenci poruszają się w dzień, natomiast spotkania odbywają się wieczorami,*
- *każdego dnia agent musi zmienić miejsce pobytu,*
- *agenci mogą poruszać się jedynie po drogach łączących miasta (niestety dodatkowym utrudnieniem jest fakt, iż w Bajtocji drogi są jednokierunkowe),*
- *agent nie może jednak poruszać się zbyt szybko (mogło by to wzbudzać niepotrzebne zainteresowanie) — jednego dnia nie może przemieścić się dalej niż do sąsiedniego miasta,*
- *odległość między dwoma miastami połączonymi drogą jest na tyle mała, że agent wyruszający z pierwszego miasta zawsze dotrze do drugiego miasta przed wieczorem,*
- *spotkanie uznaje się za odbyte w momencie, gdy dwaj agenci znajdą się tego samego wieczora w tym samym mieście.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `AGE.IN` *liczbę miast i opis sieci dróg Bajtocji, oraz pozycje początkowe dwóch agentów,*
- *stwierdzi, czy możliwe jest ich bezpieczne spotkanie, a jeśli tak, to po ilu dniach,*
- *zapisze wynik w pliku tekstowym* `AGE.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `AGE.IN` *znajdują się dwie liczby całkowite n i m, oddzielone pojedynczym odstępem, gdzie $1 \leqslant n \leqslant 250$, $0 \leqslant m \leqslant n \cdot (n-1)$.*

W drugim wierszu znajdują się dwie liczby całkowite a_1 i a_2 oddzielone pojedynczym odstępem, $1 \leqslant a_1, a_2 \leqslant n$ oraz $a_1 \neq a_2$, oznaczające, odpowiednio, początkowe pozycje agentów nr 1 i nr 2.

W m następnych wierszach znajdują się pary liczb naturalnych a i b oddzielone pojedynczymi odstępami, $1 \leqslant a, b \leqslant n$ oraz $a \neq b$, oznaczające istnienie drogi z miasta a do miasta b.

Wyjście

Plik tekstowy `AGE.OUT` *powinien zawierać dokładnie 1 wiersz zawierający:*

- *dokładnie jedną dodatnią liczbę całkowitą t, oznaczającą minimalny czas (w dniach) potrzebny do zorganizowania bezpiecznego spotkania dwóch agentów – jeżeli do takiego spotkania można doprowadzić,*
- *słowo* `NIE`*, gdy nie można doprowadzić do bezpiecznego spotkania.*

Przykład

Dla pliku wejściowego `AGE.IN`*:*

```
6 7
1 5
1 2
4 5
2 3
3 4
4 1
5 4
5 6
```

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

```
3
```

ROZWIĄZANIE

Sieć dróg Bajtocji reprezentujemy przez graf G o n wierzchołkach (miastach) i m krawędziach (drogach). Pierwszym, najbardziej narzucającym się rozwiązaniem, jest symulacja poruszania się obu agentów. Oznaczmy przez $s = (i, j)$ stan obliczeń oznaczający, iż w tym samym czasie pierwszy agent może znajdować się w mieście o numerze i, a drugi w mieście o numerze j. Teraz rozważmy graf G' zbudowany na stanach s, ma on n^2 wierzchołków i m^2 krawędzi. W przypadku posługiwania się grafem G' problem sprowadza się do znalezienia najkrótszej ścieżki z wyróżnionego wierzchołka $s_0 = (a_1, a_2)$ do dowolnego wierzchołka postaci (i, i). Ponieważ wszystkie krawędzie mają wagę 1, więc wystarczy przejść graf G' wszerz. Nie trzeba konstruować grafu G' wprost, można także, jak w poniższym przykładzie, dynamicznie generować jego krawędzie.

Schemat algorytmu (Q i $Q2$ oznaczają kolejki typu "pierwszy wchodzi-pierwszy wychodzi", czyli FIFO):

```
Q:={a_1,a_2};
{ początkowo, dla dowolnego i, j: v[i,j]=0 }
v[a_1,a_2]:=1;
t:=0;
while not Q.pusta do begin
  Q2:=pusta_kolejka;
  while not Q.pusta do begin
    (u,v):=Q.PobierzElement;
    if (u=v) then
        "Znaleziono rozwiązanie — t"
        STOP;
    for i∈ {k : (u,k) jest krawędzią w G} do
      for j∈ {k : (v,k) jest krawędzią w G} do
        if v[i,j]=0 then begin
          v[i,j]:=1;
          Q2.Dodaj(i,j);
        end
  end;
  t:=t+1;
  Q:=Q2;
end;
"Brak rozwiązania"
```

Teraz zastanówmy się jaką złożoność ma powyższy algorytm. Oznaczmy przez $K(i, j)$ koszt jaki trzeba ponieść aby wygenerować następniki stanu (i, j). Wówczas $K(i, j) = d_i \cdot d_j$, gdzie d_l oznacza liczbę krawędzi wychodzących z wierzchołka l. Ponieważ w pesymistycznym przypadku trzeba będzie rozpatrzyć wszystkie stany, całkowity koszt może wynosić:

$$n^2 + \sum_{i=1..n}\sum_{j=1..n} K(i,j) = n^2 + \sum_{i=1..n}\sum_{j=1..n} d_i \cdot d_j = n^2 + \sum_{i=1..n}\left(d_i \cdot \sum_{j=1..n} d_j\right) =$$

$$= n^2 + \sum_{i=1..n}(d_i \cdot m) = n^2 + m \cdot \sum_{i=1..n} d_i = n^2 + m^2$$

Łączny koszt to $O(n^2 + m^2)$.

Rozwiązanie wzorcowe polega na pewnym ulepszeniu poprzedniego rozwiązania. W poprzednim rozwiązaniu jedna faza algorytmu polegała na wykonaniu ruchu oboma agentami. Tym razem podzielimy tą fazę na dwie części: ruch pierwszym agentem i ruch drugim agentem. Co prawda takie rozwiązanie dwukrotnie powiększa przestrzeń stanów: $s = (i,j)$ (ten sam stan co w poprzednim rozwiązaniu), oraz dochodzą nowe stany $s' = (i',j')$, oznaczające, że pierwszy agent wykonał już swój ruch i znajduje się na pozycji i', natomiast drugi powinien teraz wykonać ruch, a na razie znajduje się na pozycji j'. Jednak dzięki tej metodzie znajdowanie następników stanów jest nieco mniej kosztowne, dla stanów $s = (i,j)$ potrzeba d_i operacji, natomiast dla stanów $s' = (i',j')$ potrzeba $d_{j'}$ operacji. Tak jak w poprzednim algorytmie, dotarcie do stanu $s = (i,i)$ oznacza, że agenci mogą zorganizować bezpieczne spotkanie.

Teraz już możemy naszkicować schemat algorytmu (Q i $Q2$ oznaczają kolejki typu "pierwszy wchodzi-pierwszy wychodzi", czyli FIFO):

```
Q:={a_1,a_2};
{ początkowo, dla dowolnego i, j: v[i,j]=0, oraz v'[i,j]=0 }
v[a_1,a_2]:=1;
t:=0;
while not Q.pusta do begin
  Q2:=pusta_kolejka;
  {ruch pierwszego agenta}
  while not Q.pusta do begin
    (u,v):=Q.PobierzElement;
    if (u=v) then
       "Znaleziono rozwiązanie — t"
       STOP;
    for i∈ {k : (u,k) jest krawędzią w G} do
      if v'[i,v]=0 then begin
        v'[i,v]:=1;
        Q2.Dodaj(i,v)
      end
  end;
  {ruch drugiego agenta}
  while not Q2.pusta do begin
    (u,v):=Q2.PobierzElement;
    for i∈ {k : (v,k) jest krawędzią w G} do
      if v[u,i]=0 then begin
        v[u,i]:=1;
        Q.Dodaj(i,v)
      end
  end;
  t:=t+1
end;
"Brak rozwiązania"
```

Analiza złożoności

Niech $K(i,j)$ oznacza koszt poniesiony na znalezienie następników stanu $s = (i,j)$. Tak więc $K(i,j) = d_i$. Analogicznie $K'(i,j)$ oznacza koszt dla stanów typu $s' = (i',j')$, a zatem $K'(i,j) = d_j$. W pesymistycznym przypadku łączny koszt całego algorytmu wynosi:

$$2n^2 + \sum_{i=1..n}\sum_{j=1..n} K(i,j) + \sum_{i=1..n}\sum_{j=1..n} K'(i,j) = 2n^2 + \sum_{i=1..n}\sum_{j=1..n} d_i + \sum_{i=1..n}\sum_{j=1..n} d_j =$$

$$= 2n^2 + \sum_{i=1..n} n \cdot d_i + \sum_{i=1..n} m = 2n^2 + n \cdot \sum_{i=1..n} d_i + n \cdot m = 2n^2 + 2nm$$

Więc to rozwiązanie ma złożoność $O(nm)$, co daje w pesymistycznym przypadku (dla grafów mających $O(n^2)$ krawędzi) $O(n^3)$ operacji.

Inne rozwiązania

Innym rozwiązaniem, mającym złożoność $O(dm)$, gdzie d oznacza minimalny czas potrzebny do zorganizowania bezpiecznego spotkania (lub n^2, jeśli takiego spotkania nie można zorganizować), jest obliczanie do jakich miast mogą dojść w i-tym kroku agenci nr 1 i 2. Jeśli te dwa zbiory mają część wspólną, oznacza to możliwość spotkania w i-tym dniu.

Niestety w niektórych przypadkach czas potrzebny do zoorganizowania spotkania może być bardzo duży, nawet rzędu n^2. Tak więc w pesymistycznych przypadkach to rozwiązanie jest równie wolne jak pierwsze.

(Zmienne $V1$ i $V2$ oznaczają zbiory)

```
V1:={a_1};
V2:={a_2};
dl:=0;
while (not V1.pusty and not V2.pusty and dl < n^2) do
begin
  dl:=dl+1;
  jeśli istnieje wierzchołek v należący do V1 i V2
     to "Znaleziono rozwiązanie";
  V1:=wierzchołki v takie, że istnieje wierzchołek u ∈ V1
        i (u,v) należy do G;
  V2:=wierzchołki v takie, że istnieje wierzchołek u ∈ V2
        i (u,v) należy do G
end;
"Brak rozwiązania"
```

TESTY

Do sprawdzania rozwiązań zawodników użyto 16 testów:

Testy oznaczone $K_2(b_1, b_2, n)$ oznaczają testy składające się z dwóch pełnych grafów skierowanych (o rozmiarach b_1, b_2) połączonych ścieżką długości n

Testy oznaczone $E(n_1, n_2, b_1, b_2)$ oznaczają testy składające się z dwóch pełnych grafów skierowanych (o rozmiarach b_1, b_2) połączone dwoma ścieżkami o wspólnym końcu, różniącymi się długością o 1. Na ścieżkach tych występują dodatkowo cykle długości n_1, n_2.

- AGE0.IN — test z treści zadania,
- AGE1.IN — mały graf z odpowiedzią negatywną,

- AGE2.IN — mały graf z odpowiedzią pozytywną,
- AGE3.IN — $K_2(30, 30, 10)$,
- AGE4.IN — $K_2(50, 50, 50)$,
- AGE5.IN — $K_2(100, 100, 50)$,
- AGE6.IN — $K_2(75, 75, 100)$,
- AGE7.IN — $K_2(50, 50, 150)$,
- AGE8.IN — $E(51, 61, 134, 0)$,
- AGE9.IN — $E(53, 59, 124, 10)$,
- AGE10.IN — $E(51, 59, 109, 20)$,
- AGE11.IN — $E(53, 64, 80, 30)$,
- AGE12.IN — duży graf dwudzielny, $n = 180$, z odpowiedzią negatywną,
- AGE13.IN — duży graf dwudzielny, $n = 250$, agenci znajdują się w tej samej części grafu,
- AGE14.IN — duży graf dwudzielny, $n = 150$, dołączone przejście gwarantujące rozwiązanie,
- AGE15.IN — duży graf dwudzielny, $n = 250$, z odpowiedzią negatywną.

Testy 11 i 12 oraz 14 i 15 zostały zgrupowane.

Adam Malinowski
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Powtórzenia

Dany jest ciąg słów nad alfabetem $\{'a', \ldots, 'z'\}$. Należy znaleźć długość najdłuższego słowa występującego jako spójny fragment w każdym z danych słów.

Zadanie

Napisz program, który:

- *wczyta ciąg słów z pliku tekstowego* `POW.IN`*,*
- *obliczy długość najdłuższego słowa występującego jako spójny fragment w każdym z podanych słów,*
- *zapisze wynik w pliku tekstowym* `POW.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `POW.IN` *zapisano liczbę n, gdzie $1 \leqslant n \leqslant 5$, oznaczającą liczbę słów. W każdym z n kolejnych wierszy znajduje się jedno słowo utworzone z małych liter alfabetu angielskiego $'a', \ldots, 'z'$. Każde ze słów ma długość przynajmniej 1, ale nie większą niż 2 000.*

Wyjście

Plik tekstowy `POW.OUT` *powinien zawierać dokładnie jeden wiersz zawierający pojedynczą liczbę całkowitą równą długości najdłuższego słowa występującego jako spójny fragment w każdym z danych słów.*

Przykład

Dla pliku wejściowego `POW.IN`*:*

```
3
abcb
bca
acbc
```

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

```
2
```

ROZWIĄZANIE

Do rozwiązania zadania posłuży nam adaptacja struktury danych zwanej *słownikiem podsłów bazowych*. Słownik podsłów bazowych to z grubsza rzecz biorąc tablica *nazw* wszystkich podsłów danego słowa o długościach będących potęgami dwójki, przy czym takie same podsłowa mają te same nazwy, a różne podsłowa o tej samej długości — różne nazwy (nazwa to po prostu liczba całkowita z zakresu od 1 do długości słowa; `num`$[i,k]$ to nazwa podsłowa o długości 2^k zaczynającego się na pozycji i w danym słowie). Taka informacja umożliwia nam sprawdzenie w czasie stałym, czy dwa podsłowa tej samej długości są jednakowe, nawet jeśli ich długości nie są potęgami dwójki: nazwą identyfikującą jednoznacznie podsłowo długości r ($2^k < r < 2^{k+1}$) zaczynające się na pozycji i jest para $\langle \mathtt{num}[i,k], \mathtt{num}[i+r-2^k+1,k] \rangle$. Rozmiar tej struktury danych, to oczywiście $\Theta(d)$, gdzie d jest długością słowa, i można ją skonstruować w takim właśnie czasie *metodą wstępującą* (ang. bottom-up):

- Najpierw, na podstawie kodów poszczególnych znaków, nadajemy nazwy słowom jednoliterowym.

- Jeśli nadaliśmy już nazwy wszystkim podsłowom długości 2^k (czyli wypełniliśmy wiersz $\texttt{num}[k,*]$ w naszej tablicy), to teraz podsłowu długości 2^{k+1} zaczynającemu się na pozycji i, gdzie $1 \leqslant i \leqslant d - 2^{k+1} + 1$, nadajemy tymczasową nazwę $\langle \texttt{num}[i,k], \texttt{num}[i+2^k,k] \rangle$. Jest to dobra, unikalna nazwa — tyle, że nie jest liczbą z zakresu od 1 do d, lecz parą takich liczb.

- Zamieniamy teraz nazwy będące parami na nazwy liczbowe. W tym celu sortujemy wszystkie pary leksykograficznie (kubełkowo, zaczynając od mniej znaczącej współrzędnej, zob. [7], [13]) i wykonujemy przenumerowanie (takie same wektory będą po posortowaniu leżały koło siebie). Nazwy po przenumerowaniu umieszczamy w kolejnych kolumnach tablicy `num` (oczywiście wraz z każdym wektorem musimy przechowywać numer pozycji początku odpowiadającego mu podsłowa).

Czas wypełniania każdego wiersza jest liniowy, a liczba wierszy do wypełnienia to $\lceil \log d \rceil$. Szczegóły konstrukcji można znaleźć w książce [10].

W rozwiązaniu wzorcowym obliczamy *wspólny* słownik dla wszystkich zadanych słów, sklejonych w jedno długie słowo z separatorami (specjalnymi znakami występującymi tylko jednokrotnie) na granicach słów wejściowych. Separatory wprowadzamy po to, żeby uniknąć odrębnego analizowania podsłów „przechodzących przez granice" słów wejściowych. Żeby zaoszczędzić pamięć, przechowujemy tylko jeden, ostatnio obliczony wiersz tablicy `num`. Schemat postępowania wygląda następująco:

- Obliczamy nazwy podsłów o długościach będących kolejnymi potęgami dwójki. Robimy to dopóty, dopóki istnieje podsłowo długości 2^k występujące w każdym ze słów wejściowych.

- Kiedy już wiemy, że najdłuższe podsłowo wspólne dla wszystkich słów wejściowych ma długość mieszczącą się w zakresie $2^k, \ldots, 2^{k+1} - 1$, to jej dokładną wartość wyznaczamy metodą bisekcji w k krokach. (Zachodzi tu potrzeba wyznaczenia nazw dla podsłów o długości p na podstawie nazw dla podsłów o długości r, gdzie $r < p < 2r$. Robimy to, posługując się opisaną na początku sztuczką, czyli jako nazwę tymczasową biorąc parę nazw zachodzących na siebie podsłów długości r pokrywających podsłowo długości p.)

Inne rozwiązania

Przedstawione tu rozwiązanie nie jest optymalne. Postawiony w zadaniu problem można rozwiązać w czasie *liniowym* względem sumy długości słów wejściowych. Wymaga to jednak użycia dość wyrafinowanych struktur danych (np. *drzew sufiksowych*), raczej trudnych do zaimplementowania podczas pięciogodzinnej sesji. W dodatku stopień komplikacji tych struktur mógłby spowodować na tyle duże stałe narzuty czasowe, że przy określonych w zadaniu ograniczeniach na rozmiar danych wejściowych, rozwiązanie asymptotycznie optymalne byłoby przy testowaniu trudne do odróżnienia lub wręcz gorsze, od znacznie prostszego rozwiązania opisanego powyżej.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 14 testów, niekiedy łączonych w grupy.

- POW0.IN — test z treści zadania,
- POW1.IN — prosty test poprawnościowy,
- POW2.IN — test poprawnościowy, słowa składające się z róznych liter,
- POW3.IN — test poprawnościowy, trzy identyczne słowa,
- POW4.IN — prosty test poprawnościowy, 4 krótkie słowa losowe,
- POW5.IN — test poprawnościowy, trzy kolejne słowa Fibonacciego,
- POW6.IN — test poprawnościowy, 1 krótkie i 4 długie słowa,

- POW7.IN — test wydajnościowy, 3 słowa o długości 100,
- POW8.IN — test wydajnościowy, 5 słów o długości 400,
- POW9.IN — test wydajnościowy, 3 słowa o długości 400,
- POW10.IN — test wydajnościowy, 4 słowa o długości 1000,
- POW11.IN — test wydajnościowy, 2 słowa o długości 1200,
- POW12.IN — test wydajnościowy, 5 słów o długości 2000,
- POW13.IN — test wydajnościowy, 5 słów o długości 2000.

Tomasz Waleń — autor zadania, opracowanie
Marcin Sawicki — program wzorcowy

Promocja

Wielka bajtocka sieć supermarketów poprosiła Cię o napisanie programu symulującego koszty właśnie przygotowywanej promocji.

Przygotowywana promocja ma mieć następujące zasady:

- *klient, który chce wziąć udział w promocji, wpisuje na zapłaconym przez siebie rachunku swoje dane i wrzuca go do specjalnej urny,*
- *pod koniec każdego dnia promocji z urny wyciągane są dwa rachunki:*
  - *najpierw wybierany jest rachunek opiewający na największą kwotę,*
  - *następnie wybierany jest rachunek opiewający na najmniejszą kwotę;*

  klient, który zapłacił największy rachunek otrzymuje nagrodę pieniężną równą różnicy pomiędzy wysokością jego rachunku, a wysokością rachunku opiewającego na najmniejszą kwotę,
- *aby uniknąć kilkukrotnych nagród za jeden zakup, oba wybrane wg reguł z poprzedniego punktu rachunki nie wracają już do urny, ale wszystkie pozostałe rachunki dalej biorą udział w promocji.*

Obroty supermarketu są bardzo duże, możesz więc założyć, że pod koniec każdego dnia, przed wyciągnięciem rachunków opiewających na największą i najmniejszą kwotę, w urnie znajdują się co najmniej 2 rachunki.

Twoim zadaniem jest obliczenie na podstawie informacji o wysokościach rachunków wrzucanych do urny w poszczególnych dniach promocji, jaki będzie łączny koszt nagród w całej promocji.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `PRO.IN` *listę wysokości rachunków wrzucanych do urny w poszczególnych dniach promocji,*
- *obliczy łączny koszt nagród wypłacanych w kolejnych dniach promocji,*
- *zapisze wynik w pliku tekstowym* `PRO.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `PRO.IN` *znajduje się jedna dodatnia liczba całkowita n, gdzie $1 \leqslant n \leqslant 5000$, oznaczająca czas trwania promocji w dniach.*

W każdym z kolejnych n wierszy znajduje się ciąg nieujemnych liczb całkowitych pooddzielanych pojedynczymi odstępami. Liczby w $(i+1)$-szym wierszu pliku określają wysokości rachunków wrzuconych do urny w i-tym dniu promocji. Pierwsza w wierszu liczba k, $0 \leqslant k \leqslant 10^5$, jest liczbą rachunków z danego dnia, a kolejne k liczb to dodatnie liczby całkowite będące wysokościami poszczególnych rachunków, każda z tych liczb jest nie większa niż 10^6.

Łączna liczba rachunków wrzuconych do urny podczas całej promocji nie przekracza 10^6.

Wyjście

Plik tekstowy `PRO.OUT` *powinien zawierać dokładnie jedną liczbę całkowitą równą łącznemu kosztowi nagród wypłacanych podczas całej promocji.*

Przykład

Dla pliku wejściowego `PRO.IN`*:*

```
5
```

```
3 1 2 3
2 1 1
4 10 5 5 1
0
1 2
```

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

```
19
```

ROZWIĄZANIE

Przy rozwiązaniu tego zadania bardzo pomocna będzie struktura danych umożliwiająca wykonywanie następujących operacji:

`ADD(x)` – dodanie elementu `x` do struktury;

`MIN` – podanie wartości minimalnego elementu w strukturze;

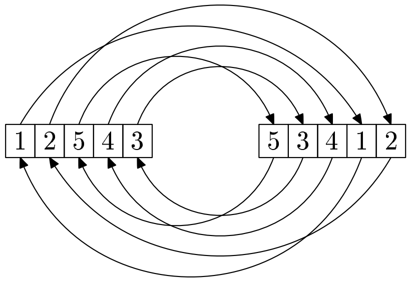
`MAX` – podanie wartości maksymalnego element w strukturze;

`ExtractMin` – usunięcie minimalnego elementu;

`ExtractMax` – usunięcie maksymalnego elementu.

Zwykły kopiec (zob. [13]) oferuje efektywną implementację trzech z tych operacji: `ADD, MIN, ExtractMin` (lub `ADD, MAX, ExtractMax`). Rozwiązaniem może być struktura składającą się z dwóch połączonych kopców, jeden z nich udostępnia operacje `ADD, MIN, ExtractMin`, natomiast drugi `ADD, MAX, ExtractMax`. W takim rozwiązaniu decydujemy się na pewną rozrzutność i każdy element należący do struktury będziemy pamiętać podwójnie: w pierwszym i drugim kopcu, dodatkowo z każdym elementem jest związany wskaźnik do bliźniaczego elementu w drugim kopcu. Operacje `MIN, MAX` odwołują się do odpowiednich kopców i wymagają stałego czasu. Operacja `ADD` wymaga dodania elementu do obu kopców, oraz uaktualnienia wskaźników, i tak jak w zwykłym kopcu wymaga czasu $O(logn)$. Operacje `ExtractMin, ExtractMax` sprowadzają się do wykonania odpowiedniej operacji na odpowiednim kopcu (tym który oferuje tę operację), oraz usunięciu elementu bliźniaczego z drugiego kopca (ponieważ pamiętamy wskaźnik do tego elementu, możemy wykonać i tę operację w czasie $O(logn)$). Zatem całość również wymaga czasu $O(logn)$. Przy wszystkich modyfikacjach, należy zwracać baczną uwagę na uaktualnianie wskaźników między oboma kopcami.

Rys. 1 Przykład podwójnego kopca



Rozwiązanie 1

Mając do dyspozycji powyższą stukturę danych, schemat algorytmu jest już oczywisty. Dla każdego dnia dodajemy do struktury rachunki, a następnie wyjmujemy z niej jeden największy i jeden najmniejszy, oraz odpowiednio zwiększamy całkowity koszt promocji.

Niestety takie rozwiązanie, ze względu na bardzo dużą liczbę rachunków, które mogły pojawić się w promocji, nie jest akceptowalne ze względu na ogranicznia pamięciowe. Należy zastanowić się, jakie rachunki na pewno nie mają szans na wyjęcie z urny. Można spostrzec, że gdy mamy w strukturze więcej niż $2n$ rachunków, to szanse na wyciągnięcie z urny ma jedynie n największych i n najmniejszych, natomiast wszystkie pozostałe, "środkowe" rachunki, nie mają już takiej szansy i można je pominąć. To spostrzeżenie prowadzi już do ulepszonego rozwiązania, które będzie wymagać jedynie pamięci rzędu $O(n)$.

Rozwiązanie 2

- Dopóki liczba rachunków w strukturze S nie przekracza $2n$ stosuj rozwiązanie 1.
- Jeśli liczba rachunków osiągnie $2n$, przepisz n najmiejszych do struktury S_{min}, natomiast n największych do struktury S_{max}. Strukturę S można już usunąc, natomiast wszystkie dalsze operację będą wykonywane na S_{min} i S_{max}. Dodawanie nowego rachunku x wygląda następująco:
  - jeśli $x < S_{min}.MAX$, to usuwamy z S_{min} element $S_{min}.MAX$ (operacja $S_{min}.ExtractMax$) i dodajemy x do S_{min};
  - jeśli $x > S_{max}.MIN$, to postępujemy symetrycznie jak w poprzednim przypadku (zamieniając MAX na MIN i S_{min} na S_{max});
  - w przeciwnym przypadku możemy zapomnieć o tym rachunku.

Chcąc wydobyć rachunek o najmniejszej wartości wykonujemy operację $ExtractMin$ na S_{min} i analogicznie, chcąc otrzymać rachunek o największej wartości wykonujemy operację $ExtractMax$ na S_{max}.

TESTY

Do sprawdzania rozwiązań zawodników użyto 10 testów opisanych poniżej:

- PRO1-6.IN — proste testy poprawnościowe,
- PRO7.IN — trudny test wydajnościowy, $n = 5000$, $K = 100006$,
- PRO8.IN — prosty test, prawie każdego dnia dwa rachunki, $n = 5000$, $K = 210004$,
- PRO9.IN — duży test wydajnościowy, $n = 5000$, $K = 1000000$,
- PRO10.IN — duży, losowy test wydajnościowy, $n = 5000$, $K = 999800$.