Zawody II stopnia

opracowania zadaĆń

Grzegorz Jakacki **Albert Krzymowski**
treść zadania, opracowanie program wzorcowy

Addon

Addon, nowo odkryty pierwiastek promieniotwórczy jest najwydajniejszym ze znanych paliw jądrowych. Trwają prace nad uruchomieniem energetycznego reaktora addonowego.
Projekt przewiduje, że komora paliwowa reaktora będzie miała postać pionowej rury. W komorze, jeden na drugim będą ustawione pręty paliwa, czyli walce z addonu. Pręty paliwowe będą produkowane w różnych długościach.
Cykl pracy reaktora rozpoczyna się od wstawienia paliwa do komór. Kolejnym krokiem jest zapłon paliwa. Niestety, wysokość słupa paliwa biorącego udział w reakcji nie może być dowolna, ponieważ tylko dla niektórych wysokości reakcja przebiega bezpiecznie. Wysokości te nazywamy **wysokościami stabilnymi**.
Projektanci reaktora mają dwa zadania: ustalić wysokość komory paliwowej oraz dobrać zestaw długości, w jakich będą produkowane pręty addonu.
Mówimy, że zestaw długości jest **bezpieczny** *dla danej komory, jeżeli wysokość dowolnego słupa, jaki można ustawić w komorze z prętów o długościach z tego zestawu, jest stabilna.*
Mówimy, że zestaw długości jest **pełny** *dla danej komory, jeżeli z prętów o długościach z tego zestawu można ustawić każdy słup o stabilnej wysokości nie większej niż wysokość tej komory.*

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego ADD.IN zbiór wysokości stabilnych,*
- *oblicza maksymalną wysokość komory, dla której istnieje zestaw długości jednocześnie bezpieczny i pełny,*
- *znajduje, dla takiej komory, zestaw długości bezpieczny i pełny mający minimalną liczbę elementów,*
- *zapisuje wyniki w pliku tekstowym ADD.OUT*

Wejście

W pierwszym wierszu pliku tekstowego ADD.IN jest zapisana liczba naturalna $1 \leqslant n \leqslant 10000$. Jest to liczba danych wysokości stabilnych.
W każdym z n kolejnych wierszy jest zapisana jedna liczba całkowita dodatnia nie większa niż 1000. Są to dane wysokości stabilne zapisane w porządku rosnącym.

Wyjście

W pierwszym wierszu pliku ADD.OUT należy zapisać jedną liczbę — maksymalną wysokość komory.
W kolejnych wierszach — rosnący ciąg liczb (każdą w osobnym wierszu) stanowiący wyznaczony zestaw długości.

Przykład

Dla pliku ADD.IN

```
14
5
10
12
15
17
20
21
22
24
26
27
```

```
30
31
33
```

Poprawnym rozwiązaniem jest plik ADD.OUT

```
24
5
12
21
```

Twój program powinien szukać pliku ADD.IN w katalogu bieżącym i tworzyć plik ADD.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę ADD.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku ADD.EXE.

ROZWIĄZANIE

Liczbą **generowalną** ze zbioru liczbowego A będziemy nazywać liczbę, która jest elementem A lub daje się przedstawić jako suma pewnych elementów A (być może z powtórzeniami). Np. 11 jest liczbą generowalną (lub „generuje się”) ze zbioru $\{2, 5, 7, 12\}$. Zbiór wszystkich liczb generowalnych ze zbioru A będziemy oznaczać symbolem $\mathrm{Domk}(A)^*$ (taki zbiór albo jest pusty, albo nieskończony). Trudno nie zauważyć, że w wyniku ustawiania prętów addonu jeden na drugim otrzymujemy walec o wysokości równej sumie ich długości, zatem:

Fakt 1: Wysokości słupów addonu, jakie można ustawić korzystając z prętów o długościach ze zbioru P, tworzą zbiór $\mathrm{Domk}(P)$.

Rzut oka na własności operacji Domk pozwala zauważyć następujące fakty:

Fakt 2: Jeżeli $A \subseteq B$, to $\mathrm{Domk}(A) \subseteq \mathrm{Domk}(B)$, czyli dołożenie elementów do A nie zmniejsza zbioru $\mathrm{Domk}(A)$.

Fakt 3: $A \subseteq \mathrm{Domk}(A)$.

Fakt 4: $\mathrm{Domk}(A) = \mathrm{Domk}(\mathrm{Domk}(A))$.

Fakt 5: Jeżeli $A \subseteq \mathrm{Domk}(B)$, to $\mathrm{Domk}(A) \subseteq \mathrm{Domk}(B)$.

Czytelnik z pewnością poradzi sobie z ich dowodami; dowody faktów 2 i 3 są proste, nieco pracy wymaga dowód faktu 4, fakt 5 wynika z faktów 2, 3 i 4. Zauważmy jeszcze, że:

Fakt 6: Zbiór wysokości stabilnych mniejszych od m jest pełny dla komory o wysokości m (choć nie musi być bezpieczny, jak np. $\{2, 4\}$ dla $m = 7$).

Pierwszym pojawiającym się problemem jest wyznaczenie maksymalnej wysokości komory, dla której istnieje bezpieczny i pełny zestaw prętów. Mając na uwadze jedynie wyznaczenie wysokości komory nie musimy (na razie) dbać o to, by zestaw długości prętów był minimalny (jeżeli istnieje jakikolwiek zestaw, to istnieje również minimalny). Oznaczmy przez WS zbiór **wysokości stabilnych**.

Twierdzenie 7: Niech d będzie najmniejszą liczbą nie należącą do zbioru wysokości stabilnych, ale generowalną z niego ($d = \min\{v : v \in \mathrm{Domk}(WS) \text{ oraz } v \notin WS\}$). Wtedy $w = d - 1$ jest maksymalną wysokością komory, dla której istnieje bezpieczny i pełny zbiór długości prętów.

Dowód: Dowód przeprowadzimy w dwóch fazach. Pokażemy, że dla komory o wysokości większej niż w nie istnieje bezpieczny i pełny zestaw długości prętów, a następnie pokażemy, że taki zestaw istnieje dla komory o wysokości w.

Przypuśćmy, że wysokość w nie jest maksymalna, czyli że istnieje komora o wysokości $w' > w$ spełniająca warunki zadania. Pokażemy, że z dowolnego pełnego zbioru długości prętów, dla komory o

* Jest to tzw. **domknięcie** zbioru A ze względu na dodawanie.

wysokości w', można ustawić słup wysokości d, czyli że dla takiej komory dowolny zbiór pełny nie będzie bezpieczny.

Dla dowolnego pełnego (dla komory o wysokości w') zbioru długości prętów P każda wysokość stabilna mniejsza lub równa w' musi generować się z P (z definicji pełności), czyli*

$$WS \cap [1..w'] \subseteq \mathrm{Domk}(P) \tag{1}$$

zatem wszystko, co da się wygenerować ze zbioru wysokości stabilnych nie większych od w', musi się także dać wygenerować ze zbioru P (fakt 5):

$$\mathrm{Domk}(WS \cap [0..w']) \subseteq \mathrm{Domk}(P) \tag{2}$$

Wiadomo też, że do ułożenia słupa wysokości d wystarczą pręty ze zbioru WS i to nie wszystkie, bo tylko te krótsze od d, czyli (pamiętamy, że $[1..(d-1)] = [1..w]$):

$$d \in \mathrm{Domk}(WS \cap [0..w]) \tag{3}$$

Poza tym, z tego, że $[1..w] \subseteq [1..w']$ i faktu 2 wiemy, że wszystko, co da się wygenerować ze zbioru wysokości stabilnych nie większych od w, da się także wygenerować ze zbioru wysokości stabilnych nie większych od w':

$$\mathrm{Domk}(WS \cap [1..w]) \subseteq \mathrm{Domk}(WS \cap [1..w']) \tag{4}$$

Z (3), (4) i (2) otrzymamy:

$$d \in \mathrm{Domk}(WS \cap [1..w]) \subseteq \mathrm{Domk}(WS \cap [1..w']) \subseteq \mathrm{Domk}(P)$$

czyli ostatecznie $d \in \mathrm{Domk}(P)$, co oznacza, że przy ustaleniu wysokości komory w' większej niż w, z prętów o długościach z dowolnego pełnego zbioru P można ustawić słup o wysokości d, który mieści się w tej komorze, choć wysokość d nie jest stabilna. Zatem dla komory o wysokości większej niż w żaden zbiór pełny nie jest bezpieczny.

Pokażemy teraz, że dla komory o wysokości w istnieje bezpieczny i pełny zbiór długości prętów P. Wystarczy wziąć zbiór wszystkich wysokości stabilnych mniejszych lub równych w, czyli $P = WS \cap [1..w]$. Ten zbiór jest pełny (fakt 6).

Dla dowodu własności bezpieczeństwa P zauważmy, że każda liczba, która generuje się z P, generuje się także z WS, czyli $\mathrm{Domk}(P) \subseteq \mathrm{Domk}(WS)$ (bo $P = WS \cap [1..w] \subseteq WS$). Zatem, jeżeli z P generuje się liczba p nie będąca wysokością stabilną ($p \in \mathrm{Domk}(P)$ i $p \notin WS$), to generuje się ona także z WS, czyli

$$p \in \mathrm{Domk}(WS) \quad \text{oraz} \quad p \notin WS$$

Zauważmy, że d miało być najmniejszą liczbą o tej własności, więc $p \geqslant d$. Stąd otrzymujemy $p > w$ (bo $w = d - 1$). Zatem jeżeli z prętów o długościach z P ustawimy słup o wysokości nie będącej wysokością stabilną, to słup ten nie zmieści się w komorze o wysokości w. □

W celu znalezienia liczby d będziemy konstruowali kolejne (coraz większe) elementy zbioru $\mathrm{Domk}(WS)$ i porównywali je z elementami zbioru WS (zapisanymi w pliku wejściowym w kolejności rosnącej). Na potrzeby opisu algorytmu potraktujmy plik wejściowy jak tablicę we : **array**$[1..n]$ **of integer** — ułatwi to jego zapis. Niech zmienna k przechowuje wartość wskaźnika pozycji w pliku, tj. numer pierwszego nie przeczytanego rekordu pliku (na początku $k = 1$).

Na zmiennej *nast* będzie przechowywana najmniejsza liczba generowalna ze zbioru $we[1..(k-1)]$ nie będąca jego elementem:

$$nast = \min \big\{ w : w \in \mathrm{Domk}\big(we[1..(k-1)]\big) \text{ oraz } w \notin we[1..(k-1)] \big\} \tag{5}$$

* Dowód łatwiej jest czytać, gdy pamięta się, że $WS \cap [1..w]$ oznacza zbiór tych wysokości stabilnych, które nie przekraczają wysokości komory w.

Rys. 1 Schemat algorytmu znajdującego maksymalną wysokość komory

$nast := 2 \cdot we[1]$;
$k := 2$;
$koniec :=$ **false**;
while $(k \leqslant n)$ **and not** $koniec$ **do** (1)
  if $we[k] > nast$ **then** $koniec :=$ **true**
  else begin
    $popraw(nast)$;
    $k := k + 1$
  end;
end;
$w := nast - 1$;

Liczba ta jest „kandydatem" na kolejną wysokość stabilną w tym sensie, że jeżeli wysokość stabilna odczytana z pliku okaże się większa od *nast*, to znaczy, że *nast* jest pierwszą zabronioną wysokością komory (liczba d) — zob. schemat algorytmu na rys. 1.

Teraz musimy jeszcze powiedzieć, w jaki sposób zadbać o to, aby był zachowany niezmiennik (5), czyli napisać procedurę oznaczoną na rys. 1 symbolem *popraw*. W tym miejscu można sformułować naiwny algorytm przeglądający za każdym razem sumy wszystkich par ze zbioru $we[1..(k-1)]$, co daje złożoność jednego obrotu pętli (1) rzędu k^2, a stąd całego algorytmu $O(n^3)$. Spróbujmy zrobić to lepiej, konstruując przy okazji minimalny pełny i bezpieczny zbiór długości prętów P.

Sztuka polega na tym, by prócz konstruowanego zbioru długości prętów przechowywać dodatkowo zbiór „kandydatów" na wartość *nast*.

Oznaczmy przez P konstruowany zbiór długości prętów, a przez *Kand* — zbiór „kandydatów".

Na początku do zbioru P wrzucamy pierwszą (najmniejszą) wysokość stabilną. Zauważmy, że jeżeli w obrocie pętli (1) wartość $we[k] < nast$ to znaczy, że $we[k]$ nie generuje się ze zbioru $we[1..(k-1)]$. Co więcej, w takim przypadku $we[k]$ nie generuje się nawet z całego zbioru *WS* (wynika to z własności dodawania liczb dodatnich: suma jest zawsze większa od składników). Przypadek $we[k] = nast$ oznacza zaś, że $we[k]$ generuje się z $we[1..(k-1)]$, zatem $we[k]$ na pewno nie należy wkładać do P.

Tajemnicą pozostaje jednak w dalszym ciągu zbiór *Kand* i sposób uaktualniania wartości zmiennej *nast*.

Spójrzmy jeszcze raz na równanie (5) będące definicją wartości *nast*. Załóżmy, że dla zbioru $we[1..(k-1)]$ zbudowaliśmy zbiór P długości prętów. Z faktów, że $we[1..(k-1)]$ jest pełny dla wysokości $we[k-1]$ oraz że każdy element generowalny z tego zbioru jest też generowalny z P wynika, że *nast* musi mieć postać $v + p$, gdzie $v \in we[1..(k-1)]$ i $p \in P$. Liczby tej postaci będziemy właśnie przechowywać w zbiorze *Kand*. Wartość *nast* otrzymamy znajdując minimum tego zbioru.

Z każdym obrotem pętli będziemy starali się utrzymać warunki (niezmienniki):

$$Kand \cup \{nast\} = \{v + p : v \in we[1..k-1], p \in P\}$$
$$nast = \min\{v + p : v \in we[1..k-1], p \in P\}$$

Na rys. 2 przedstawiono schemat algorytmu wykorzystującego powyższe mechanizmy.

Pozostaje jeszcze kwestia efektywnej implementacji operacji na zbiorach, w tym operacji *extractMin* powodującej wyjęcie ze zbioru elementu o najmniejszej wartości.

Operacje na zbiorze P nie sprawiają w zasadzie problemu, gdyż wykonujemy tylko inicjowanie zbioru, dopisywanie nowego elementu oraz przeglądanie wszystkich jego elementów. Najprostsze wydaje się trzymanie elementów zbioru P kolejno w tablicy. Koszt operacji wstawienia przy takiej implementacji wynosi $O(1)$, przejrzenie zbioru $O(n)$.

Zastosowanie podobnej implementacji dla zbioru *Kand* nie przyniesie jednak oczekiwanych rezultatów. Problemem jest operacja *extractMin*, która przy takiej implementacji może wymagać przejrzenia całego zbioru *Kand*, który może być rozmiaru rzędu n^2.

Rozwiązaniem powyższego problemu jest implementacja zbioru *Kand* za pomocą kopca. Kopiec jest strukturą danych umożliwiającą wykonywanie operacji wstawiania elementu i usuwania najmniejszego

Rys. 2 Schemat algorytmu znajdującego wysokość komory i zestaw prętów

$nast := 2 \cdot we[1];$
$k := 2;$
$Kand := \varnothing;$
$P := \{we[1]\};$
$koniec :=$ **false**;
while $(k \leqslant n)$ **and** (**not** $koniec$) **do**
  if $we[k] > nast$ **then** $koniec :=$ **true**
  else begin
    if $we[k] < nast$ **then** $P := P \cup we[k];$ (1)
    $Kand := Kand \cup \{we[k] + p : p \in P\};$ (2)
    if $we[k] = nast$ **then** $nast := extractMin(Kand);$ (3)
    $k := k + 1$
  end;
end;
$w := nast - 1;$

elementu w czasie proporcjonalnym do logarytmu rozmiaru kopca. Przy takiej implementacji otrzymujemy następujące oszacowanie na czas działania algorytmu:*

$$n \cdot (\underbrace{O(1)}_{\text{wiersz (1)}} + \underbrace{O(n \log n^2)}_{\text{wiersz (2)}} + \underbrace{O(\log n^2)}_{\text{wiersz (3)}}) = O(n^2 \log n)$$

W niniejszym opracowaniu nie będziemy wdawać się w szczegóły implementacji kopca. Zainteresowanych odsyłamy do pozycji [7].

Wydawało by się, że powyższego oszacowania nie da się już poprawić. Można jednak skorzystać z pewnej „technicznej” sztuczki;** po pierwsze, zakres wartości, jakie przechowujemy w kopcu jest na tyle niewielki (dokładnie [1..10000]), że kopiec można zastąpić tablicą o rozmiarze 10000 bitów, w której ustawienie bitu n na 1 będzie oznaczało, że „liczba n jest w kopcu”. Wstawienie elementu do takiej struktury polega na prostym wyliczeniu adresu i wykonaniu przypisania, zatem czas sumaryczny takiej operacji jest stały.

Co z operacją *extractMin*? Można za każdym razem przeszukiwać tablicę od początku i znajdować najmniejszy jej element. Sumaryczna liczba odczytów tablicy podczas pracy programu wynosiłaby około $10000^2/2$. Liczbę tę można zredukować do około 10000, gdy zauważymy, że po wyjęciu elementu *nast* ze struktury *Kand*, nie trafiają już do niej elementy mniejsze od *nast*. Zatem poszukiwania nie trzeba każdorazowo rozpoczynać od początku tablicy — wystarczy szukać od miejsca, w którym zakończono poprzednie przeszukiwanie.

W pliku ADD.PAS na załączonej dyskietce przedstawiono rozwiązanie wzorcowe z użyciem kopców. Przeróbkę implementacji struktur danych na tablice bitowe pozostawiamy jako ćwiczenie dla Czytelnika.

OPIS TESTÓW

Do sprawdzenia rozwiązań zawodników użyto 10 testów (ADD0.IN–ADD9.IN).

- ADD0.IN — test z treści zadania.
- ADD1.IN–ADD5.IN — testy poprawnościowe (liczby wysokości stabilnych: 1, 9, 45, 65, 27).
- ADD6.IN–ADD9.IN — testy wydajnościowe (liczby wysokości stabilnych: 900, 702, 950, 9980).

* Oczywiście $O(\log n^2) = O(\log n)$.

** Sztuczka przestałaby działać, gdyby np. zrezygnowano z założenia, że wysokości są liczbami całkowitymi.

Przemysława Kanarek
treść zadania

Marcin Kubica
program wzorcowy, opracowanie

Genotypy

Genotypy są skończonymi łańcuchami genów. Opisujemy je za pomocą słów utworzonych z wielkich liter alfabetu angielskiego A–Z. Różne litery oznaczają różne rodzaje genów. Gen może pączkować — zmieniając się w dwójkę nowych genów. Tymi przemianami rządzi skończony zbiór reguł. Każdą regułę pączkowania można zapisać w postaci trójki wielkich liter $A_1 A_2 A_3$, co oznacza, że gen A_1 może się zmienić w dwójkę genów $A_2 A_3$.
Wielką literą S oznaczamy specjalny rodzaj genów zwanych supergenami. Hodowla genotypu rozpoczyna się od łańcucha supergenów i polega na sterowanym pączkowaniu wybranych genów zgodnie z ustalonymi regułami.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego GEN.IN skończony zbiór reguł pączkowania genów oraz ciąg słów określających genotypy, jakie należy wyhodować,*
- *dla każdego z danych genotypów bada, czy można go wyhodować z pewnego skończonego łańcucha supergenów zgodnie z danymi regułami pączkowania i jeśli tak, znajduje minimalną długość takiego łańcucha,*
- *zapisuje wyniki w pliku tekstowym GEN.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego GEN.IN jest zapisana jedna liczba całkowita $1 \leqslant n \leqslant 10000$. W każdym z n kolejnych wierszy jest zapisana jedna reguła pączkowania, w postaci słowa złożonego z trzech wielkich liter A–Z. W następnym wierszu jest zapisana jedna liczba całkowita $1 \leqslant k \leqslant 10000$. W każdym z k kolejnych wierszy jest zapisany jeden genotyp w postaci niepustego słowa złożonego z co najwyżej 100 liter A–Z.

Wyjście

W i-tym z kolejnych k wierszy pliku tekstowego GEN.OUT należy zapisać:

- *jedną liczbę całkowitą dodatnią oznaczającą minimalną długość łańcucha supergenów potrzebnego do wyhodowania i-tego danego genotypu, albo*
- *jedno słowo NIE, jeśli tego genotypu nie da się wyhodować.*

Przykład

Dla pliku GEN.IN:

```
6
SAB
SBC
SAA
ACA
BCC
CBC
3
ABBCAAABCA
CCC
BA
```

poprawnym rozwiązaniem jest plik GEN.OUT:

```
3
1
```

```
NIE
```

Twój program powinien szukać pliku GEN.IN w katalogu bieżącym i tworzyć plik GEN.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę GEN.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku GEN.EXE.

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

Dopuszczalne są reguły pączkowania, w których na drugim lub trzecim miejscu występuje supergen.

ROZWIĄZANIE

Zadanie to jest adaptacją znanego problemu informatycznego: jak sprawdzić czy dane słowo należy do języka generowanego przez gramatykę bezkontekstową (zob. książka [15]). Adaptacja polega na częściowym uproszczeniu problemu. Zamiast dwóch rodzajów symboli, terminalnych i nieterminalnych, są tylko jednoliterowe nazwy genów. Supergen S jest odpowiednikiem aksjomatu. Reguły pączkowania genów są uproszczonym odpowiednikiem produkcji — po prawej stronie znajdują się zawsze dwa symbole. Problem należenia danego słowa do języka generowanego przez gramatykę bezkontekstową można sformułować następująco: czy dany genotyp można uzyskać z pojedynczego supergenu. Różnica w liczbie supergenów od których zaczynamy zniknie, jeśli najpierw spróbujemy rozwiązać trochę bardziej ogólny problem.

Niech dany genotyp ma postać $G = g_1 \dots g_l$. Dla $1 \leqslant i \leqslant j \leqslant l$, niech $W(i,j)$ będzie zbiorem takich pojedynczych genów, z których można uzyskać genotyp $g_i \dots g_j$. W szczególności, jeśli $\mathrm{S} \in W(1,l)$ to cały genotyp G można uzyskać z jednego supergenu.

Zauważmy, że dla $i = 1, \dots, l$, $W(i,i) = \{g_i\}$. Ponadto, jeśli pewien gen A może wypączkować tworząc parę genów BC oraz (dla $1 \leqslant i \leqslant k < j \leqslant l$) z genu B możemy uzyskać genotyp $g_i \dots g_k$, a z genu C możemy uzyskać genotyp $g_{k+1} \dots g_j$, to oczywiście z genu A możemy uzyskać genotyp $g_i \dots g_j$. Oznaczmy przez *Sklej* następującą operację na zbiorach genów:

$$Sklej(V_1, V_2) = \bigcup_{v_1 \in V_1, v_2 \in V_2} \{g : g v_1 v_2 \text{ jest jedną z reguł pączkowania genów}\}$$

Wówczas dla $i < j$:

$$W(i,j) = \bigcup_{i<k\leq j} Sklej(W(i,k-1), W(k,j))$$

Zbiory $W(i,j)$ możemy wyliczyć za pomocą programowania dynamicznego, w dwuwymiarowej tablicy W : **array** $[1..l,\ 1..l]$ **of set of** *gen*. Najpierw inicjalizujemy przekątną tablicy jednoelementowymi zbiorami $\{g_i\}$, a następnie kolejne nadprzekątne wypełniamy korzystając z operacji *Sklej*. Ilustruje to poniższy fragment programu*.

```
for i := 1 to l do
  W[i,i] := {g_i};
for i := 1 to l − 1 do
  for j := 1 to l − i do begin
    W[j,i+j] := ∅;
    for k := 1 to i do
      W[j,i+j] := W[j,i+j] ∪ Sklej(W[j,j+k−1],W[j+k,i+j])
  end;
```

* Program, którego fragmenty tu przytaczamy, można znaleźć na dyskietce, w pliku GEN.PAS

Wymaga to wykonania $\Theta(l^3)$ operacji *Sklej*. Ich implementacja jest więc kluczowa dla efektywności rozwiązania. Ponieważ w każdym pliku z danymi zestaw reguł pączkowania jest ten sam dla wszystkich genotypów, można go wstępnie przygotować na potrzeby operacji *Sklej*.

Ze względu na małą liczbę rodzajów genów, zbiory genów można reprezentować za pomocą liczb typu `longint`, przeznaczając na każdy gen jeden bit. Reguły pączkowania genów możemy zapamiętać w dwuwymiarowej tablicy zbiorów genów P : **array** [*gen*, *gen*] **of longint** — $P[g_1, g_2]$ to zbiór genów, z których można wypączkować parę genów $g_1 g_2$. Funkcja *Sklej* może mieć następującą implementację:

```
function Sklej (V1, V2 : longint) : longint;
var
  g1, g2 : gen;
  zb : longint;
begin
  zb := 0;
  for g1 ∈ gen do
    if g1 ∈ V1 then
      for g2 ∈ gen do
        if g2 ∈ V2 then
          zb := zb or P[g1, g2];
  Sklej := zb
end;
```

Mając wyliczoną tablicę W możemy wyznaczyć liczbę potrzebnych supergenów — ponownie stosując programowanie dynamiczne. Wypełniamy jednowymiarową tablicę S:**array** $[1..l]$ **of integer** tak, aby $S[i]$ było równe minimalnej liczbie supergenów potrzebnych do uzyskania genotypu $g_1 \ldots g_i$. Jeśli supergen należy do zbioru $W[1, i]$ to oczywiście $S[i] = 1$. W przeciwnym przypadku szukamy takiego podziału genotypu $g_1 \ldots g_i$ na dwie części $g_1 \ldots g_{k-1}$, $g_k \ldots g_i$, że pierwszą z nich można uzyskać z jak najmniejszej liczby supergenów, a drugą z jednego supergenu. Ilustruje to poniższy fragment programu.

```
for i := 1 to l do begin
  if supergen ∈ W[1, i] then
    S[i] := 1
  else begin
    S[i] := ∞;
    for k := 2 to i do
      if supergen ∈ W[k, i] then
        S[i] := min(S[i], S[k − 1] + 1);
  end
end;
```

Wynik znajduje się w $S[l]$, przy czym jeśli $S[l] = \infty$, to danego genotypu nie da się uzyskać.

Koszt czasowy tej części programu wynosi $\Theta(l^2)$, jest więc zaniedbywalny w porównaniu z kosztem wyliczenia tablicy W.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 15 testów GEN0.IN – GEN14.IN. Poniżej zamieszczamy krótkie komentarze do testów.

- GEN1.IN — z jednego supergenu można uzyskać takie genotypy złożone z genów A i B, w których jest tyle samo genów A i B. Test ten bada ogólną poprawność rozwiązania;
- GEN2.IN — z jednego supergenu można uzyskać genotypy przypominające wyrażenia nawiasowe. Pierwszy i ostatni gen tworzą parę nawiasów: C...D lub A...Z. W środku znajduje się jeden gen

X lub para genotypów przypominających wyrażenia nawiasowe. Test ten bada ogólną poprawność rozwiązania;

- GEN3.IN — jeśli geny A–J zastąpimy cyframi 0–9, to z jednego supergenu można uzyskać genotypy będące zapisami liczb podzielnych przez 3 zakończonymi pojedynczym supergenem. Test ten bada ogólną poprawność rozwiązania;
- GEN4.IN — z jednego supergenu można uzyskać genotypy postaci

$$\mathrm{A}\ldots\mathrm{AB}\ldots\mathrm{BC}\ldots\mathrm{C}$$

  w których występuje przynajmniej jeden gen A, liczba genów B jest dowolna, a liczba genów C jest równa liczbie genów A plus podwojona liczba genów B. Test ten bada ogólną poprawność rozwiązania;
- GEN5.IN — z jednego supergenu można uzyskać takie genotypy złożone z genów A, B i C, w których liczba genów A jest równa liczbie genów B plus podwojona liczba genów C. Test ten eliminuje rozwiązania badające rekurencyjnie jakie genotypy można uzyskać z jednego supergenu;
- GEN6.IN — z jednego supergenu można uzyskać ciąg 4-ech lub 7-miu genów A. Ile supergenów potrzeba, aby uzyskać ciąg 67-miu genów A? Test ten eliminuje rozwiązania badające rekurencyjnie, które fragmenty genotypu powinny powstać z pojedynczych supergenów;
- GEN7.IN, GEN8.IN — w testach tych geny reprezentują pewne zbiory. Reguły pączkowania pokrywają wszystkie takie sytuacje, w których suma zbiorów reprezentowanych przez geny powstałe w pączkowaniu zawiera się w zbiorze reprezentowanym przez gen poddawany pączkowaniu. Testy te zawierają dużo reguł pączkowania i badają efektywność implementacji operacji *Sklej*;
- GEN9.IN, GEN10.IN — w testach tych geny reprezentują pewne ciągi. Reguły pączkowania pokrywają wszystkie takie sytuacje, w których sklejenie ciągów reprezentowanych przez geny powstałe w pączkowaniu daje podciąg ciągu reprezentowanego przez gen poddawany pączkowaniu. Testy te zawierają dużo reguł pączkowania i badają efektywność implementacji operacji *Sklej*;
- GEN11.IN–GEN14.IN — w testach tych z jednego supergenu można uzyskać takie genotypy złożone z genów A i B, w których jest tyle samo genów A i B. Testy te zawierają coraz dłuższe genotypy i badają wydajność rozwiązań.

Wojciech Guzicki
treść zadania, opracowanie

Marcin Engel
program wzorcowy

Lotniska

W państwie X istnieją lotniska w n miastach. Znane są maksymalne **przepustowości** *tych lotnisk — lotnisko w mieście M_i może mieć co najwyżej d_i połączeń lotniczych z innymi miastami. Należy zaplanować sieć połączeń lotniczych między tymi miastami w taki sposób, by miasto M_i miało dokładnie d_i połączeń z innymi miastami, przy czym zakładamy, że każde połączenie jest dwukierunkowe i każde miasto może mieć z innym tylko jedno połączenie.*

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego LOT.IN liczbę n miast oraz liczby d_i,*
- *układa sieć połączeń lotniczych, taką że dla każdego i od 1 do n miasto M_i ma dokładnie d_i połączeń z innymi miastami,*
- *zapisuje w pliku LOT.OUT listę wszystkich tworzących sieć połączeń.*

Dane są tak dobrane, by rozwiązanie zadania istniało. Jeśli zadanie ma wiele rozwiązań, Twój program powinien znajdować tylko jedno.

Wejście

W pierwszym wierszu pliku tekstowego LOT.IN jest zapisana liczba całkowita n spełniająca nierówność $3 \leqslant n \leqslant 500$. Jest to liczba miast.
W kolejnych n wierszach są zapisane liczby całkowite dodatnie d_i, po jednej w każdym wierszu.

Wyjście

W pliku tekstowym LOT.OUT należy zapisać wszystkie połączenia lotnicze sieci utworzonej przez Twój program. Każde połączenie należy zapisać w osobnym wierszu w postaci dwóch liczb całkowitych dodatnich oddzielonych pojedynczym odstępem, tj. numerów dwóch połączonych miast. Numery miast w wierszu mogą występować w dowolnej kolejności; również kolejność zapisywania połączeń w pliku jest dowolna.

Przykład

Dla pliku LOT.IN

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

przykładem poprawnego rozwiązania jest plik LOT.OUT

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

Twój program powinien szukać pliku LOT.IN w katalogu bieżącym i tworzyć plik LOT.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę LOT.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku LOT.EXE.

UZUPEŁNIENIE TREŚCI

W trakcie trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

> *Może się zdarzyć, że podróż z jednego miasta do drugiego, nawet z przesiadkami, nie jest możliwa.*

ROZWIĄZANIE

Liczbę d_i połączeń lotniska o numerze i z innymi lotniskami nazwiemy *przepustowością* tego lotniska. Szukaną sieć połączeń znajdujemy w następujący sposób:

(1) Znajdujemy lotnisko mające największą przepustowość. Przypuśćmy, że jego numerem jest i.

(2) Łączymy lotnisko i z d_i lotniskami o największych przepustowościach spośród pozostałych lotnisk. Przepustowości tych lotnisk zmniejszamy o 1.

(3) Lotnisko o numerze i usuwamy z listy lotnisk. Podobnie usuwamy wszystkie lotniska mające (po ostatnim zmniejszeniu) przepustowość równą 0.

(4) Powtarzamy kroki (1)–(3) dotąd, aż usuniemy wszystkie lotniska.

Poprawność tej procedury wynika łatwo przez indukcję względem n. Łatwo sprawdzić, że podstawa indukcji zachodzi dla $n = 3$. Załóżmy, że opisany wyżej sposób postępowania prowadzi do rozwiązania zadania, gdy liczba lotnisk jest mniejsza od n i przypuśćmy, że mamy do czynienia z n lotniskami. Bez zmniejszenia ogólności można założyć, że lotniska zostały ponumerowane w ten sposób, by przepustowości kolejnych lotnisk tworzyły ciąg nierosnący: $d_1 \geqslant d_2 \geqslant d_3 \geqslant \ldots \geqslant d_n$. Opisany wyżej algorytm polega na wybraniu lotniska o numerze 1 i połączeniu go z lotniskami o numerach $2, 3, \ldots, d_1 + 1$. Wykażemy, że jeśli istnieje jakakolwiek sieć połączeń, w której lotnisko i ma d_i połączeń z innymi lotniskami, to istnieje również sieć taka, w której lotnisko i ma d_i połączeń z innymi lotniskami i ponadto lotnisko 1 jest połączone z lotniskami $2, 3, \ldots, d_1 + 1$. Stąd wynika, że po wykonaniu kroków (1)–(3) nadal możemy otrzymać rozwiązanie zadania. Liczba lotnisk uległa jednak zmniejszeniu co najmniej o 1 (bo usunęliśmy lotnisko 1 z listy), więc powtarzanie kroków (1)–(3) doprowadzi do rozwiązania. To kończy dowód indukcyjny.

Pozostaje dowieść, że istnieje rozwiązanie, w którym lotnisko 1 jest połączone z lotniskami $2, 3, \ldots, d_1 + 1$. Weźmy pod uwagę jakąkolwiek sieć połączeń spełniającą warunki zadania. Jeśli lotnisko 1 jest połączone z lotniskami $2, 3, \ldots, d_1 + 1$, to dowód jest zakończony. Przypuśćmy więc, że lotnisko 1 nie jest połączone z którymś z tych lotnisk. Niech i będzie najmniejszym numerem lotniska, które nie jest połączone z lotniskiem 1. Oczywiście $i \leqslant d_1 + 1$. Ale lotnisko 1 jest połączone z d_1 lotniskami, musi więc być połączone z jakimś lotniskiem o numerze większym od $d_1 + 1$. Niech $j > d_1 + 1$ i niech j będzie numerem lotniska, z którym połączone jest lotnisko nr 1. Ponieważ przepustowości lotnisk są uporządkowane nierosnąco, więc $d_i \geqslant d_j$. To znaczy, że lotnisko i ma co najmniej tyle połączeń, co lotnisko j. Lotnisko j jest połączone z lotniskiem 1, z którym nie jest połączone lotnisko i. Stąd wynika, że istnieje lotnisko, na przykład o numerze k, które jest połączone z lotniskiem i i nie jest połączone z lotniskiem j. Mamy więc cztery lotniska: 1, i, j, k takie, że i jest połączone z k oraz 1 z j, ale ani 1 nie jest połączone z i, ani j z k. Teraz zmieniamy połączenia: lotnisko i zamiast z k łączymy z 1, lotnisko j zamiast z 1 łączymy z k. Zauważamy, że stopnie lotnisk nie uległy zmianie, natomiast lotnisko 1 jest już połączone z lotniskiem i. W ten sposób powiększyliśmy o jedno liczbę połączeń lotniska 1 z lotniskami o numerach $2, 3, \ldots, d_1 + 1$. Powtarzając to postępowanie wielokrotnie (ścisłe rozumowanie znów wymaga indukcji), doprowadzimy do sytuacji, w której lotnisko 1 będzie połączone z d_1 następnymi lotniskami. To ostatecznie kończy dowód poprawności algorytmu.

Może się zdarzyć, że za pomocą opisanej wyżej procedury stworzymy sieć w której z niektórych lotnisk nie będzie można dolecieć (nawet z przesiadkami) do niektórych innych. Można pokazać, że jeśli suma stopni lotnisk jest równa co najmniej $2n - 2$, to istnieje sieć „spójna”, tzn. taka, w której każde lotnisko ma połączenie (niekoniecznie bezpośrednie) ze wszystkimi innymi lotniskami. Główny pomysł dowodu polega na tym, że jeśli $d_1 + \ldots + d_n \geqslant 2n - 2$, to sieć połączeń albo jest spójna, albo ma cykl. Jeśli połączone ze sobą lotniska x i y nie mają żadnych połączeń z lotniskami pewnego cyklu, to wybieramy z tego cyklu dwa sąsiednie lotniska u i v, a następnie zmieniamy połączenia: łączymy x z u i y z v. W ten sposób zmniejszamy o 1 liczbę tzw. składowych, tzn. spójnych części całej sieci. Powtarzając to postępowanie odpowiednią liczbę razy otrzymamy w końcu sieć spójną.

Jedna z możliwych realizacji zadania polega na dokładnym powtórzeniu powyższego rozumowania. Umieszczamy w tablicy pary: numer lotniska i jego stopień, a następnie sortujemy nierosnąco względem stopni. Następnie bierzemy kolejne lotniska (o numerach Nr od 1 do n) i jeśli stopień lotniska jest większy od zera, to zmniejszamy stopnie następnych d_{Nr} lotnisk o 1. Teraz musimy ponownie posortować tablicę. To sortowanie nie jest trudne, zamiany wymagają tylko dwa położone obok siebie bloki wierzchołków. Znalezione połączenia od razu zapisujemy w pliku wyjściowym.

Należy zauważyć, że jeśli w tablicy lotnisk występują kolejno lotniska mające ten sam stopień, to w czasie zmniejszania stopni możliwe są dwa przypadki:

- wszystkie lotniska tego bloku lotnisk będą miały zmniejszony stopień; wtedy do tego miejsca tablica jest nadal posortowana,

- zmniejszanie stopni zakończy się wewnątrz tego bloku i otrzymamy blok postaci:

$$k-1, k-1, k-1, \ldots, k-1, k, k, k, \ldots, k.$$

  Jeśli w tym bloku występuje L lotnisk mających stopień $k - 1$ i P lotnisk mających stopień k, to trzeba zamienić ze sobą dwa bloki: jeśli $L < P$, to zamieniamy L lotnisk z lewej strony z L lotniskami z prawej strony, a jeśli $L \geqslant P$, to P lotnisk z prawej strony z P lotniskami z lewej strony. Nie mamy istotnych ograniczeń pamięci, więc możemy przepisać lotniska z jednej strony do dodatkowej tablicy, wpisać do zwolnionych miejsc lotniska z drugiej strony i w końcu wpisać na właściwe miejsce lotniska z tablicy pomocniczej.

Takie rozwiązanie znajduje się w pliku `LOT1.PAS`. Sortowanie tablicy zostało dokonane za pomocą procedury sortowania szybkiego (quicksort). Rozwiązania mniej efektywne otrzymamy używając innych, wolniejszych procedur sortowania oraz zastępując opisaną wyżej zamianę bloków ponownym sortowaniem całej tablicy. Implementacja tej wersji algorytmu wraz z procedurami sortowania przez wstawianie, przez wybieranie, bąbelkowego, przez łączenie oraz sortowania szybkiego znajduje się w pliku `LOT2.PAS`.

Implementacja algorytmu z nawrotami znajduje się w pliku `LOT3.PAS`. Jest to algorytm nieefektywny zarówno czasowo, jak i pamięciowo.

Rozwiązanie wzorcowe bazuje na strukturze danych, w której numery lotnisk są zapisywane w kolejkach. Przy wczytywaniu danych tworzymy dla każdej przepustowości d kolejkę, do której wrzucamy numery lotnisk o przepustowości d. Przy okazji zapamiętujemy maksymalną przepustowość.

Powiemy, że kolejka przechowująca lotniska o przepustowości d jest rzędu d.

Załóżmy, że w pewnej chwili działania algorytmu maksymalny rząd niepustej kolejki wynosi d. Pojedynczy krok algorytmu polega na wyjęciu z kolejki rzędu d jednego lotniska. Następnie wyjmujemy z tej kolejki, a następnie w miarę ubywania elementów z kolejek o rzędach coraz mniejszych d lotnisk. Każde takie lotnisko zostaje umieszczone w kolejce o jeden rząd mniejszej niż ta, z której zostało wyjęte. Oczywiście, lotnisk wyjmowanych z kolejki rzędu 1 nie umieszczamy już w strukturze. Przy realizacji tego algorytmu należy uważać, aby nie pobierać ponownie wierzchołków, które zostały przerzucone do kolejki o mniejszym rzędzie.

Złożoność tego algorytmu jest liniowa względem liczby połączeń w konstruowanej sieci, gdyż po każdym wyjęciu lotniska z kolejki mamy „w ręku” kolejne połączenie, a operacje na kolejce mają koszt stały.

Rozwiązanie to zostało zaimplementowane w pliku `LOT.PAS`. Czasy działania programów `LOT.PAS` i `LOT1.PAS` dla przeprowadzonych testów nie odbiegały bardzo od siebie.

TESTY

Do sprawdzenia zadania zaprojektowano 9 testów. Pierwszym (LOT0.IN) był test z treści zadania. Następne 4 testy były również dość małych rozmiarów (LOT1.IN–LOT4.IN) i miały na celu sprawdzenie poprawności algorytmów. Dwa testy (LOT5.IN i LOT6.IN) były nieco większe, miały po ok. 150 lotnisk i różniły się liczbą połączeń. Ostatnie dwa testy miały odpowiednio 300 i 500 lotnisk i ich zadaniem było sprawdzenie efektywności programów.

Tomasz Śmigielski
treść zadania

Albert Krzymowski
program wzorcowy, opracowanie

Paliwo

Firma transportowa X otrzymała nowe zlecenie; będzie dostarczała paczki z miasta A do miasta B. Paczki będą przewożone samochodami firmy. Na trasie AB jest wiele stacji benzynowych oferujących paliwo po różnych cenach. Pierwsza z nich znajduje się na początku trasy. Wszystkie samochody firmy X zużywają jednakowo standardową jednostkę paliwa na przejechanie jednej mili, ale mają różne pojemności zbiorników. Koszt paliwa potrzebnego do przejechania trasy AB zależy od pojemności zbiornika na paliwo w samochodzie i wybranego planu tankowania paliwa na stacjach leżących wzdłuż trasy. Zakładamy, że na każdej stacji jest zawsze wystarczająco dużo paliwa, by kierowca mógł napełnić zbiornik do pełna oraz że leżą one dostatecznie gęsto, aby każdy samochód firmy X mógł przejechać całą trasę.

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego PAL.IN następujące dane:*
  - *pojemność zbiornika na paliwo w samochodzie,*
  - *liczbę stacji benzynowych na trasie AB,*
  - *cenę paliwa na każdej stacji oraz odległości między kolejnymi stacjami,*
- *znajduje minimalny koszt tankowania paliwa na trasie AB dla samochodu o danej pojemności zbiornika,*
- *zapisuje wynik w pliku tekstowym PAL.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego PAL.IN jest zapisana pojemność p zbiornika samochodu. Jest to liczba całkowita spełniająca nierówność $1 \leqslant p \leqslant 1000000$.
W drugim wierszu jest zapisana liczba n stacji benzynowych na trasie AB. Jest to liczba całkowita spełniająca nierówność $1 \leqslant n \leqslant 1000000$.
W każdym z kolejnych n wierszy jest zapisana para liczb całkowitych dodatnich, oddzielonych pojedynczym odstępem c_i, d_i. Liczba c_i — to cena standardowej jednostki paliwa na stacji i (licząc od A do B), zaś d_i to odległość tej stacji w milach od następnej stacji w kierunku B (d_n — to odległość ostatniej stacji od końca trasy AB). Liczby te spełniają nierówności: $1 \leqslant c_i \leqslant 1000$, $1 \leqslant d_i \leqslant 1000000$.
Długość trasy AB (suma d_i) jest nie większa niż 1000000.

Wyjście

W pliku tekstowym PAL.IN należy zapisać jedną liczbę całkowitą — minimalny koszt tankowania paliwa na trasie AB dla samochodu o danej pojemności zbiornika.

Przykład

Dla pliku PAL.IN

```
40
3
2 10
1 15
2 5
```

poprawnym rozwiązaniem jest plik PAL.OUT:

```
40
```

Twój program powinien szukać pliku PAL.IN w katalogu bieżącym i tworzyć plik PAL.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę PAL.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku PAL.EXE.

ROZWIĄZANIE

Powyższy problem można przedstawić jako podział rozważanej drogi na rozłączne odcinki o długości jednej mili i przyporządkowanie każdemu z nich ceny benzyny zużytej na jego przejechanie. Przyjmijmy następujące oznaczenia:

- $stacja(i)$ — stacja, na której zostało zatankowane paliwo zużyte na i-tej mili,
- $cena(s)$ — cena benzyny na stacji s,
- $koszt(i)$ — koszt przejechania i-tego odcinka drogi, $koszt(i) = cena(stacja(i))$,
- P — pojemność baku samochodu,
- K — koszt podróży, który wyraża się sumą:

$$K = \sum_{i=1}^{D} cena(stacja(i))$$

  gdzie D jest odległością między miastami A i B.

Nasze zadanie polega na zminimalizowaniu K.

Obserwacja 1: Dla każdego planu tankowania *stacja*, o całkowitym koszcie równym $K = \sum_{i=1}^{D} cena(stacja(i))$, istnieje „plan tankowania” $stacja'$ o koszcie $K' = \sum_{i=1}^{D} cena(stacja'(i))$ taki, że $K = K'$ oraz

$$\forall_{0 \leqslant i < j \leqslant D}\ stacja(i) \leqslant stacja'(j) \qquad (*)$$

Warunek $(*)$ mówi, że paliwo w baku jest zużywane w kolejności tankowania. Wynika z niego natychmiast, że żadna jednostka paliwa nie jest przewożona w zbiorniku samochodu przez więcej niż P mil. Innymi słowy, jest ona zużyta nie dalej niż w odległości P od miejsca zakupu.

Powyższe obserwacje pozwalają na szukanie rozwiązania optymalnego tylko wśród rozwiązań spełniających $(*)$.

Prezentowany poniżej program, wybiera, dla każdego odcinka drogi długości jeden, najtańszą stację znajdującą się nie dalej niż P (pojemność baku samochodu) w stronę miasta A. Wyszukiwanie rozpoczynamy od miasta A pamiętając dane o stacjach z ostatnich P mil. Można łatwo zauważyć, że nie musimy przechowywać danych o wszystkich stacjach z danego obszaru — wystarczy dla każdej ceny paliwa pamiętać najbliższą stację sprzedającą benzynę w tej samej ofercie. Różnych cen paliwa jest niewiele w porównaniu z potencjalną liczbą miejsc, w których można tankować.

Zastosowanie tablicy przechowującej tylko informacje o cenach paliwa pozwolą uzyskać stały koszt pamięciowy.

Tablica

$$Ceny\ :\ \mathbf{array}[TCena]\ \mathbf{of}\ \mathbf{0}..CMAX\_BAK + CMAX\_DYSTANS;$$

informuje jak daleko można dojechać korzystając z benzyny o danej cenie.

Główna pętla programu, obliczająca potrzebną ilość pieniędzy, przedstawia się następująco:

```
gotowka := 0;
mila := 0;
Readln(PlikWe, doNastepnejStacji, min);
Ceny[min] := PojBaku;
Dec(PozostaloStacji);
```

```
repeat
  min := znajdzNajtansza;
  if mila + doNastepnejStacji < Ceny[min] then
    skok := doNastepnejStacji
  else
    skok := Ceny[min] - mila;
  gotowka := gotowka + min · skok;
until not jedz(skok);
```

Funkcja *znajdzNajtanszą* wyszukuje najtańszą stację położoną nie dalej niż B od rozważanego odcinka w stronę miasta A.

```
function znajdzNajtansza : TCena;
var
  i : TCena;
begin
  i := 1;
  while Ceny[i] ⩽ mila do Inc(i);
  znajdzNajtansza := i;
end;
```

Funkcja *jedz* przesuwa samochód o zadaną liczbę mil.

```
function jedz(dystans : TDystans) : Boolean;
var
  cena : TCena;
begin
  mila := mila + dystans;
  doNastepnejStacji := doNastepnejStacji - dystans;
  if (doNastepnejStacji = 0) and (PozostaloStacji > 0) then begin
    Readln(PlikWe, doNastepnejStacji, cena);
    Ceny[cena] := mila + PojBaku;
    Dec(PozostaloStacji);
  end;
  jedz := (PozostaloStacji > 0) or (doNastepnejStacji ≠ 0)
end;
```

Można łatwo zauważyć, że poruszanie się co jedną jednostkę prowadziłoby do nieefektywnego programu. Często zdarza się, że koszt przejechania kolejnych odcinków trasy jest taki sam. Dlatego też, w przedstawionej implementacji, nie rozważamy każdego jednostkowego odcinka oddzielnie ale „przeskakujemy” do najbliższego miejsca, w którym należy uaktualnić dane (zmienna *skok*). Ważne są dwa typy sytuacji: dojeżdżamy do następnej stacji lub skończyło się paliwo o danej cenie i należy wybrać kolejne najtańsze.

Konieczność sprawdzania czy nie skończyła się benzyna o zadanej cenie ilustruje następujący przykład. Plik PAL.IN:

```
3
2
5 2
10 3
```

Przedstawiony algorytm działa w czasie $O(n)$ i zużywa pamięć o rozmiarze stałym, niezależnym od n.

Inne rozwiązanie o takich samych parametrach złożoności opiera się na założeniu, że na dowolnej stacji możemy odsprzedać zakupioną i niezużytą benzynę po cenie jej zakupu.

Analizujemy trasę przejazdu zgodnie z kierunkiem jazdy samochodu. Obserwacja 1 pozwala przyjąć, że paliwo zużywa się w takiej kolejności, w jakiej było zatankowane. Na każdej stacji:

- jeżeli posiadamy droższą benzynę niż ta na stacji, na której się zatrzymaliśmy, „odsprzedajemy ją” to bez straty;
- tankujemy do pełna.

Dla uproszczenia przyjmujemy, że w mieście B znajduje się stacja oferująca paliwo po cenie zerowej.

Powyższy algorytm jest równoważny czasowo i pamięciowo rozwiązaniu przedstawionemu jako pierwsze. Zauważmy, że sposób tankowanie zapewnia, iż paliwo w baku jest zawsze „ułożone” w kolejności od najtańszego do najdroższego. Po każdym tankowaniu mamy zatem najtańsze pokrycie dla znajdującego się przed nami odcinka drogi do następnej stacji.

Obserwacja 1 i wniosek pozwoliły stworzyć rozwiązanie optymalne, działające w stałej pamięci. Istnieją także inne, mniej efektywne rozwiązania. Na przykład rozwiązanie, które składa się z dwóch etapów. Przeglądamy drogę w kierunku od miasta B do A wyznaczając dla każdej stacji odległość do następnego tańszego miejsca tankowania, a następnie symulujemy jazdę samochodem. Na każdej stacji kupujemy tylko tyle paliwa, ile jest potrzebne aby dojechać do najbliższej tańszej. Jeżeli jest ona poza zasięgiem tankujemy do pełna.

Ponieważ musimy trzymać w pamięci całą potrzebną informację, koszt pamięciowy jest w tym przypadku liniowy, natomiast czas działania algorytmu jest asymptotycznie taki sam, jak poprzednio.

Wyszukiwanie następnej tańszej stacji można oczywiście prowadzić na bieżąco. W tym przypadku uzyskamy algorytm działający w czasie $O(n)$.

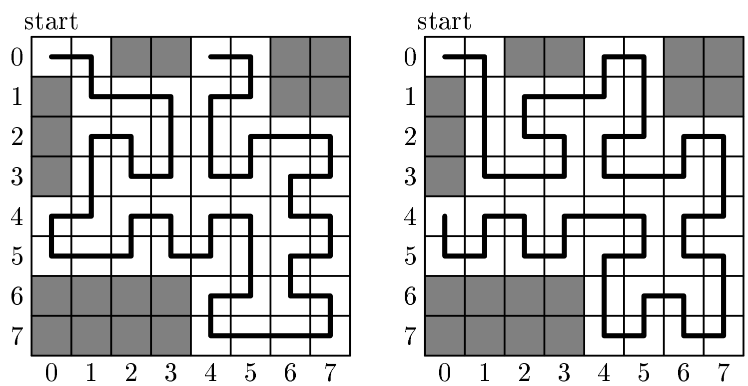
TESTY

- PAL0.IN–PAL5.IN — testy sprawdzające poprawność rozwiązania;
- PAL6.IN–PAL8.IN — testy sprawdzające szybkość działania algorytmu. Ta grupa testów eliminowała rozwiązania o złożoności czasowej $O(n^2)$;
- PAL9–PAL10.IN — testy sprawdzające zużycie pamięci. Wielkości testów zostały tak dobrane, aby wychwycić rozwiązania pamiętające dane o wszystkich stacjach.

Wojciech Rytter **Krzysztof Sobusiak**
treść zadania, opracowanie program wzorcowy

Rekurencyjna mrówka

Mrówka ma obejść wszystkie pola szachownicy $2^n \times 2^n$ poza polami zabronionymi — każde pole dokładnie jeden raz. Obchodzenie musi się zacząć w polu startowym leżącym w lewym górnym rogu i zakończyć na polu leżącym na brzegu szachownicy (mrówka na końcu opuszcza szachownicę). Zakładamy, że pole startowe nie jest zabronione. W jednym kroku mrówka może przejść do jednego z co najwyżej czterech sąsiednich pól szachownicy (w górę, dół, lewo lub prawo).
Mrówka obchodzi szachownicę w sposób w pewnym sensie rekurencyjny: aby obejść kwadrat $2^k \times 2^k$ dzieli go na cztery części (rekurencyjne ćwiartki) o rozmiarach $2^{k-1} \times 2^{k-1}$, a następnie obchodzi każdą z nich, to znaczy, jeżeli mrówka wejdzie do jednej z ćwiartek, to może z niej wyjść dopiero wtedy, gdy obejdzie wszystkie nie zabronione pola w tej ćwiartce.
Rysunek 1 przedstawia dwie trasy rekurencyjnej wędrówki mrówki na szachownicy o rozmiarach $2^3 \times 2^3$, od pola startowego (0,0) do pola leżącego odpowiednio na górnym oraz lewym brzegu szachownicy.



Rysunek 1

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego REK.IN liczbę całkowitą dodatnią n określającą jednoznacznie rozmiar szachownicy $2^n \times 2^n$ oraz liczbę m pól zabronionych i ich listę,*
- *na każdym z czterech brzegów szachownicy znajduje pole, na którym może się zakończyć rekurencyjna wędrówka mrówki, spełniająca podane wyżej warunki lub stwierdza, że takiego pola nie ma,*
- *zapisuje wyniki w pliku tekstowym REK.OUT.*

UWAGA: *Każde z czterech pól narożnych należy do dwóch brzegów szachownicy.*

Wejście

W pierwszym wierszu pliku tekstowego REK.IN jest zapisana liczba całkowita dodatnia $n \leqslant 30$.
W drugim wierszu jest zapisana liczba pól zabronionych $m \leqslant 50$.
W każdym z kolejnych m wierszy jest zapisana para liczb całkowitych nieujemnych i, j oddzielonych pojedynczym odstępem, są to dwie współrzędne: numer wiersza i oraz numer kolumny j odpowiedniego zabronionego pola. Wiersze szachownicy numerujemy od góry w dół, a kolumny od lewej do prawej, od 0 do $2^n - 1$.

Wyjście

W każdym z czterech kolejnych wierszy pliku tekstowego REK.OUT należy zapisać dwie liczby całkowite nieujemne oddzielone pojedynczym odstępem — współrzędne (numer wiersza i numer kolumny) odpowiedniego końcowego pola *rekurencyjnej trasy mrówki leżącego na: górnym, prawym, dolnym oraz lewym brzegu szachownicy (w takiej kolejności) lub jedno słowo NIE, jeśli takiego pola nie ma.*

PRZYKŁAD

Dla pliku REK.IN będącego opisem szachownicy przedstawionej na rysunku:

```
3
15
2 0
1 0
3 0
6 0
7 0
6 1
7 1
6 2
7 2
6 3
7 3
0 2
0 3
0 6
0 7
1 6
1 7
```

poprawnym rozwiązaniem jest plik REK.OUT

```
0 4
NIE
NIE
4 0
```

Twój program powinien szukać pliku REK.IN w katalogu bieżącym i tworzyć plik REK.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę REK.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku REK.EXE.

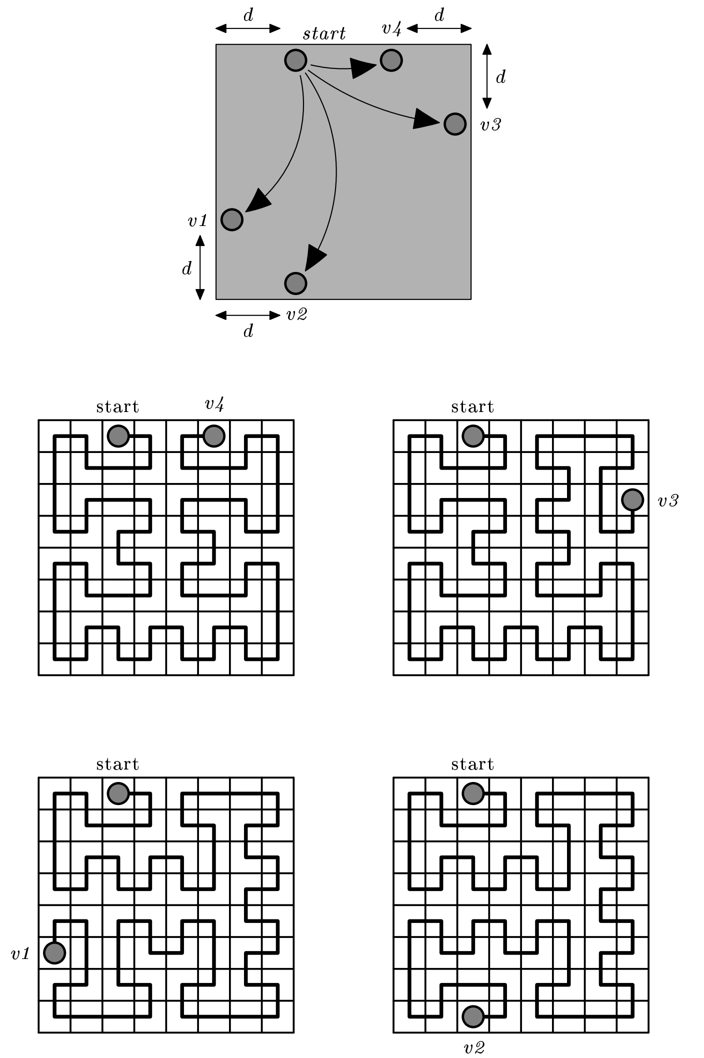
ROZWIĄZANIE

Miarę złożoności danych wejściowych określają dwie liczby: n, m, gdzie m jest liczbą zabronionych pól na szachownicy o rozmiarach $2^n \times 2^n$. Największą trudnością (oraz najbardziej istotnym punktem) w zadaniu jest potencjalnie olbrzymi rozmiar szachownicy względem n. Zadanie jest o tyle łatwe, że liczba tras zaczynających się w danym polu na zadanym boku szachownicy i kończących się na zadanym (innym lub tym samym) boku nie przekracza 1. W pewnym sensie mrówka chodzi deterministycznie, jeśli ustalimy na którym boku ma się zakończyć trasa. Trzeba wtedy zacząć chodzić zgodnie z regułami, ale czasami trzeba w jednym kroku pokonywać duże puste podszachownice (w przeciwnym wypadku trasa może być astronomicznie długa). Przez **pustą podszachownicę** rozumiemy część szachownicy bez zabronionych pól. Można łatwo wyliczyć gdzie wychodzimy z pustej ćwiartki nie obchodząc jej bezpośrednio. Możemy skorzystać z następującego faktu kombinatorycznego.

Fakt 1: Przy ustalonym polu początkowym leżącym na brzegu pustej szachownicy oraz ustalonym brzegu b, na którym mrówka ma zakończyć swoją podróż, istnieje dokładnie jedna poprawna droga rekurencyjnej mrówki, kończąca się na brzegu b (zob. rys. 1).

Dowód: Indukcja ze względu na n. Pozostawiamy go jako ćwiczenie dla czytelnika. □

Rys. 1 Schemat ogólny i konkretny przykład z $d = 2$ dla szachownicy bez zabronionych pól. Jeśli rozpoczniemy w polu *start* to możemy zakończyć jedynie w jednym z pól $v1$, $v2$, $v3$, $v4$, jeśli *start* jest polem narożnym to $v1 = v2$ oraz $v3 = v4$.



Podstawowym elementem rozwiązania jest ustalenie brzegu szachownicy na którym ma być pole końcowe. Jeśli ten brzeg ustalimy, to trasa mrówki jest wyznaczona jednoznacznie, lub może jej nie być. Mamy cztery możliwości ustalenia brzegu i w związku z tym cztery wywołania podobnego algorytmu. W dalszym tekście zakładamy, że końcowy brzeg jest ustalony.

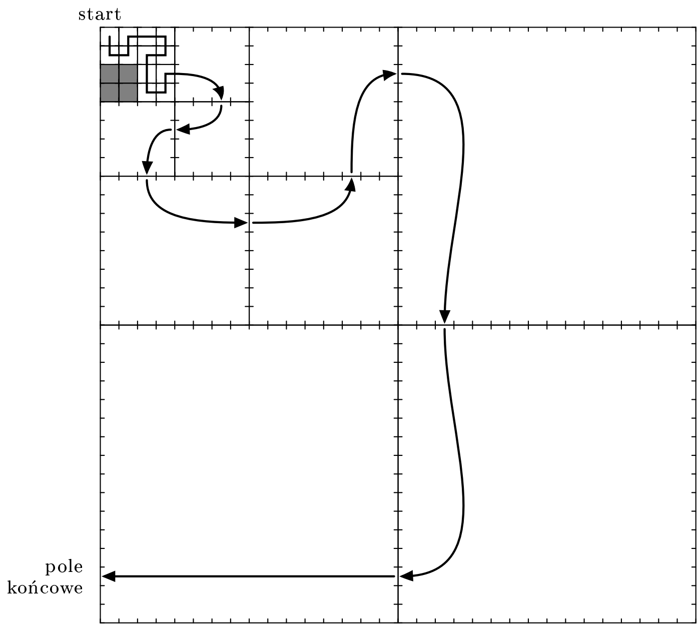
Sformułowanie zadania narzuca rozwiązanie rekurencyjne. Aby znaleźć drogę mrówki na szachownicy należy rekurencyjnie obejść jej cztery ćwiartki, za wyjątkiem tych ćwiartek, których wszystkie pola są zabronione. Kolejność przechodzenia ćwiartek jest jednoznacznie wyznaczona przez brzeg końcowy.

Przy przechodzeniu przez puste ćwiartki wykonujemy **przeskok**: w jednym kroku przechodzimy do pola końcowego na zadanym brzegu końcowym danej ćwiartki zgodnie ze schematem z rys. 1. Przykładowa strategia przeskoków jest pokazana przykładowo na rys. 2.

IMPLEMENTACJA

Podstawową częścią programu jest rekurencyjna procedura znajdowania pola końcowego mrówki dla następujących danych: pola początkowego, kierunków wejścia i wyjścia, poziomu rekurencji oraz listy pól zabronionych. Procedura jest uruchamiana czterokrotnie dla każdego z czterech brzegów szachownicy.

Rys. 2 Obliczanie końcowego pola na lewym brzegu kwadratu o rozmiarach 32×32 mającego 4 pola zabronione. Pole kończace trasę mrówki w każdym z dużych podkwadratów liczymy (w czasie stałym) zgodnie z rys. 1.



Procedura ta sprawdza najpierw, czy przekazana jej lista pól zabronionych nie jest pusta. Jeśli lista jest pusta, to oblicza współrzędne pola końcowego zgodnie rys. 1. W przeciwnym razie rozdziela przekazaną jej listę pól zabronionych zawartych w części szachownicy, dla której ma znaleźć trasę mrówki, pomiędzy jej cztery ćwiartki.

Operację tę można wykonać szybko, przez sprawdzenie dla każdego pola jednego bitu (o numerze zależnym od poziomu rekurencji) z jego współrzędnych. Procedura sprawdza też, czy któraś z ćwiartek nie jest w całości pokryta przez zabronione pola. Następnie ustala kierunek przechodzenia ćwiartek i obchodzi je rekurencyjnie, przekazując każdej swej rekurencyjnej kopii odpowiednią listę pól zabronionych.

Dla pojedynczego pola zabronionego złożoność tej procedury będzie liniowa (n wywoływań rekurencyjnych). Zatem przy m polach, pesymistycznie dostajemy czas $O(mn)$. Operacje rozdzielania list pól zabronionych amortyzują się do czasu $O(mn)$, gdyż każdy bit ze współrzędnych tych pól jest rozważany tylko raz.

UWAGA: Zakładamy, że operacje na liczbach kodujących współrzędne wykonujemy w czasie stałym.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11 testów REK0.IN–REK10.IN:

- REK0.IN — test z treści zadania;
- REK1.IN — szachownica zdegenerowana, $n = 0$;
- REK2.IN — niewielki test ($n = 3$), z czterema różnymi poprawnymi trasami mrówki;
- REK3.IN — nieduży test ($n = 4$), z jedną trasą mrówki;
- REK4.IN — mały test ($n = 4$), zawierający wszelkie dopuszczalne konfiguracje pól zabronionych w kwadracie 2×2;
- REK5.IN — test bez poprawnych tras mrówki ($n = 5$);

- REK6.IN — test z jednym polem zabronionym, bez poprawnych tras mrówki ($n = 30$);
- REK7.IN — losowy test gęsty ($n = 5$, $m = 50$);
- REK8.IN — test losowy ($n = 15$, $m = 25$);
- REK9.IN — test losowy ($n = 30$, $m = 50$);
- REK10.IN — szachownica bez pól zabronionych ($n = 30$).