Zawody II stopnia

opracowania zadań

Marcin Kubica
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Podpisy

W Urzędzie Ochrony Bajtocji (UOB) zatrudnieni są urzędnicy oraz dowódcy. W archiwum znajdują się teczki z aktami wszystkich urzędników. W każdej teczce znajduje się podpis urzędnika oraz podpisy pracowników (urzędników lub dowódców), którzy poręczają za jego lojalność. Każdy nowoprzyjmowany urzędnik musi uzyskać przynajmniej jedno poręczenie. Z biegiem czasu lista poręczycieli może się powiększać. UOB dowiedział się ostatnio, że do grona dowódców przeniknął szpieg wrogiej Mikromięklandii. Kolejni szpiedzy byli wprowadzani do UOB na stanowiska urzędnicze dzięki poręczeniu szpiega-dowódcy i/lub innych wprowadzonych szpiegów. Tacy szpiedzy mają poręczenia wyłącznie od pracowników będących szpiegami.

Wiarygodność urzędnika można podważyć, jeżeli pośrednio nie ma on poręczenia żadnego dowódcy, który nie jest szpiegiem, tzn. nie istnieje taki ciąg pracowników UOB $p_1, p_2, \ldots, p_k$*, że* p_1 *jest dowódcą nie będącym szpiegiem,* p_k *jest danym urzędnikiem i (dla* $i = 1, \ldots, k-1$*)* p_i *poręczył za* p_{i+1}*.*

Jeżeli założenie o pewnym dowódcy, że jest szpiegiem spowodowałoby, że wiarygodność urzędnika zostałaby podważona, to urzędnik ten jest podejrzany o szpiegostwo. Dowództwo UOB chciałoby zobaczyć listę takich urzędników, i to jak najszybciej!

Przykład

Dowódcy: *Anna, Grzegorz.*
Urzędnicy: *Bolesław (poręczyła Anna), Celina (poręczył Bolesław), Dorota (poręczyli Bolesław i Celina), Eugeniusz (poręczyli Anna i Grzegorz), Felicja (poręczył Eugeniusz), Halina (poręczyli Grzegorz i Ireneusz), Ireneusz (poręczyli Grzegorz i Halina).*
Podejrzani: *Bolesław, Celina, Dorota, Halina, Ireneusz.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `POD.IN` *liczbę dowódców i urzędników w UOB oraz informacje o poręczeniach,*
- *wyznaczy listę urzędników podejrzanych o szpiegostwo,*
- *wyniki zapisze do pliku tekstowego* `POD.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `POD.IN` *zapisana jest dokładnie jedna dodatnia liczba całkowita* n $(1 \leqslant n \leqslant 500)$ *będąca liczbą pracowników UOB. Pracownicy są ponumerowani od 1 do* n*. W kolejnych* n *wierszach zapisane są opisy poręczeń. W* $i+1$*-ym wierszu pliku znajduje się opis poręczeń udzielonych pracownikowi nr* i*. Jest to ciąg liczb całkowitych poodzielanych pojedynczymi odstępami. Pierwsza liczba w tym ciągu,* $0 \leqslant m_i$*, jest równa liczbie poręczeń udzielonych pracownikowi nr* i*. Kolejne* m_i *liczb to numery pracowników, którzy poręczyli za prawdomówność pracownika nr* i*. (Tak więc liczba wyrazów ciągu w* $i+1$*-ym wierszu wynosi* $m_i + 1$*.) Dowódcy to Ci pracownicy, za których nikt nie poręczył.*

Wyjście

Twój program powinien:

- *w kolejnych wierszach pliku tekstowego* `POD.OUT` *zapisać w rosnącej kolejności, po jednej liczbie w każdym wierszu, ciąg dodatnich liczb całkowitych będących numerami urzędników podejrzanych o szpiegostwo — jeżeli tacy urzędnicy są,*
- *w pierwszym i jedynym wierszu pliku wyjściowego* `POD.OUT` *zapisać dokładnie jedno słowo* `BRAK` *— jeżeli takich urzędników nie ma.*

Przykład

Dla pliku wejściowego POD.IN*:*

```
9
0
1 1
1 2
2 2 3
2 1 7
1 5
0
2 7 9
2 7 8
```

poprawną odpowiedzią jest plik tekstowy POD.OUT*:*

```
2
3
4
8
9
```

ROZWIĄZANIE

Zadanie to można w naturalny sposób wyrazić w języku teorii grafów. Mamy dany graf skierowany. Wierzchołki tego grafu reprezentują pracowników UOB, a krawędzie reprezentują poręczenia. Krawędź $v \to u$ reprezentuje poręczenie udzielone przez pracownika v urzędnikowi u. Dowódcy są reprezentowani przez te wierzchołki, do których nie prowadzą żadne krawędzie.

Niech u będzie wierzchołkiem reprezentującym urzędnika. Jak sprawdzić czy u jest podejrzany o szpiegostwo? Zależy to od tego, ilu dowódców pośrednio poręczyło za jego lojalność. Temu, że pewien dowódca d pośrednio poręczył za lojalność urzędnika u odpowiada ścieżka prowadząca z d do u. Tak więc, jeśli istnieją przynajmniej dwa wierzchołki reprezentujące dowódców, z których możemy poprowadzić ścieżki do u, to u nie jest podejrzany o szpiegostwo — gdyby jeden z tych dowódców okazał się szpiegiem, to i tak drugi poręczył pośrednio za lojalność u. Jeżeli natomiast istnieje co najwyżej jeden wierzchołek d reprezentujący dowódcę, z którego możemy poprowadzić ścieżkę kończącą się w u, to u jest podejrzany o szpiegostwo — gdyby d okazał się szpiegiem, to wiarygodność u zostałaby podważona.

Zadanie to można rozwiązać wykorzystując przeszukiwanie grafu wszerz lub w głąb (zobacz [10] lub [13]). W najprostszej wersji możemy dla każdego urzędnika wyznaczyć liczbę dowódców, którzy pośrednio poręczyli za niego, przeszukując graf "wstecz". Podobnie, możemy dla każdego dowódcy wyznaczyć urzędników, za których on poręczył, przeszukując graf zgodnie z kierunkiem krawędzi. Rozwiązania te są jednak nieefektywne. Ich złożoność czasowa jest $O(nm)$, gdzie n jest liczbą wierzchołków, a m jest liczbą krawędzi w grafie.

Zauważmy jednak, że nie musimy wyznaczać wszystkich dowódców, którzy pośrednio poręczyli za danego urzędnika. Jeśli znajdziemy dwóch takich dowódców, to nie musimy wyznaczać kolejnych — dany urzędnik i tak nie jest podejrzany o szpiegostwo. Również wszyscy ci, za których ten urzędnik poręczył też nie są podejrzani o szpiegostwo. Ponadto nie jest istotne, którzy dowódcy poręczyli pośrednio za danego urzędnika, a jedynie ilu.

Możemy więc każdemu urzędnikowi przyporządkować licznik określający ilu dowódców pośrednio poręczyło za niego: "żaden", "jeden", "dwóch lub więcej". Jeśli licznik jest równy jeden, to dodatkowo pamiętamy który dowódca poręczył pośrednio za urzędnika — informację tę będziemy wykorzystywać zamiast kolorowania wierzchołków w trakcie przeszukiwania grafu. Początkowo przyjmujemy, że żaden dowódca nie poręczył pośrednio za żadnego urzędnika. Następnie, dla każdego dowódcy przeszukujemy graf począwszy od niego i zwiększamy liczniki odwiedzonych urzędników. Jeśli licznik któregoś urzędnika jest równy jeden i poręczył za niego pośrednio ten dowódca, od którego zaczęliśmy przeszukiwanie grafu, to urzędnik ten był już w tym przeszukiwaniu odwiedzony. Jeśli licznik któregoś urzędnika osiągnie wartość "dwóch lub więcej', to pomijamy go przy dalszych przeszukiwaniach grafu. W ten sposób każdy wierzchołek zostanie odwiedzony co najwyżej dwa razy. W rezultacie złożoność czasowa takiego rozwiązania jest $O(n+m)$, gdzie n jest liczbą wierzchołków, a m jest liczbą krawędzi w grafie. Złożoność pamięciowa jest takiego rzędu jak wielkość grafu i wynosi $O(n+m)$. Rozwiązanie to może być zaimplementowane w następujący sposób:

```
const
  N_MAX = 500; { Maksymalna liczba wierzchołków. }
  ST_UNKNOWN = 0;           { Brak poręczeń. }
  { Jedno poręczenie = nr poręczającego dowódcy. }
  ST_CLEAN   = N_MAX + 1; { Co najmniej dwa poręczenia. }
type
  t_array = array [1..N_MAX] of word;
  p_array = ↑t_array;
  t_official = record
    status: word; { Licznik poręczeń. }
    cpt: boolean; { Czy jest dowódcą? }
    vouchees_no: word; { Liczba poręczanych urzędników. }
    vouchees: p_array; { Poręczani urzędnicy. }
  end
var
  officials: array [1..N_MAX] of t_official; { Graf. }
  n: word; { Liczba wierzchołków. }
procedure find_suspects;
{ Wyznacz wartości liczników poręczeń dla urzędników. }
var
  i: word;
  procedure go_down (no: word);
  { Przeszukiwanie grafu w głąb, począwszy od wierzchołka nr no. }
  var
    i: word;
    captain: word;
  begin
    captain := officials[no].status;
    for i := 1 to officials[no].vouchees_no do
      with officials[officials[no].vouchees↑[i]] do begin
        if (status = ST_UNKNOWN) then begin
          { Pierwszy raz wchodzimy do wierzchołka. }
          { Przepisujemy status poprzedniego wierzchołka. }
          status := captain;
          go_down (officials[no].vouchees↑[i])
        end else if ((status < ST_CLEAN) and
                     (status ≠ captain)) then begin
          { Znaleźliśmy drugiego dowódcę. }
          status := ST_CLEAN;
          go_down (officials[no].vouchees↑[i])
        end
      end
  end { go_down }
begin
   for i := 1 to n do officials[i].status := ST_UNKNOWN;
   for i := 1 to n do if officials[i].cpt then begin
     { Dowódca }
     officials[i].status := i;
     go_down (i);
     officials[i].status := ST_CLEAN
   end
```

end

TESTY

Do sprawdzania rozwiązań zawodników użyto 14 testów.
Większość testów ma postać: KULE(n, k, p1, p2), gdzie:

(1) cały graf n-wierzchołkowy jest podzielony na k równolicznych podgrafów

(2) każdy z podgrafów jest losowy, przy czym prawdopodobieństwo, że między danymi dwoma wierzchołkami występuje krawędź wynosi $p1$

(3) każdy podgraf zawiera dokładnie jednego dowódcę

(4) każdy podgraf zawiera jeden wyróżniony wierzchołek, który nie jest dowódcą

(5) na wierzchołkach wyróżnionych znów mamy strukturę grafu, tym razem z prawdopodobieństwem wystąpienia krawędzi równym $p2$

(6) jeśli $k = 1$, to dodatkowo doczepiamy do grafu gadżety, które powodują, że nie wszyscy urzędnicy są podejrzani (nie chcemy trywializować testu)

- POD0.IN — test z treści zadania
- POD1.IN — mały test poprawnościowy,
- POD2.IN — KULE(100, 1, 0.5),
- POD3.IN — KULE(500, 1, 0.5),
- POD4.IN — KULE(500, 1, 0.8),
- POD5.IN — KULE(500, 1, 1.0) - duży graf pełny z gadżetami,
- POD6.IN — KULE(50, 5, 0.5, 0.2),
- POD7.IN — KULE(200, 10, 0.8, 0.1),
- POD8.IN — KULE(500, 5, 0.9, 0.2),
- POD9.IN — KULE(500, 10, 1.0, 0.1),
- POD10.IN — KULE(500, 5, 1.0, 0.3),
- POD11.IN — KULE(500, 5, 0.8, 0.3),
- POD12.IN — KULE(500, 2, 0.9, 0.8),
- POD13.IN — KULE(480, 3, 0.8, 0.5).

Grzegorz Jakacki — autor zadania

Krzysztof Sobusiak — opracowanie, program wzorcowy

Automorfizmy

Turniejem *nazywamy graf skierowany, w którym:*

- *dla dowolnych dwóch różnych wierzchołków u i v istnieje dokładnie jedna krawędź pomiędzy tymi wierzchołkami (tzn. albo $u \to v$, albo $v \to u$),*
- *nie istnieją pętle (tzn. dla dowolnego wierzchołka u nie ma krawędzi $u \to u$).*

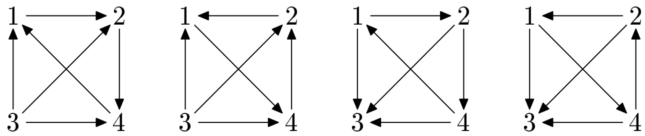
Oznaczmy przez p dowolną permutację zbioru wierzchołków turnieju. (Permutacją skończonego zbioru X nazywamy każdą różnowartościową funkcję z X w X.) Permutację p nazywamy **automorfizmem**, *jeżeli dla dowolnych dwóch różnych wierzchołków u i v zwrot krawędzi pomiędzy u i v jest taki sam, jak pomiędzy $p(u)$ i $p(v)$ (tzn. $u \to v$ jest krawędzią w turnieju wtedy i tylko wtedy, gdy $p(u) \to p(v)$ jest krawędzią w tym turnieju). Dla zadanej permutacji p interesuje nas, dla ilu turniejów jest ona automorfizmem.*

Przykład

Weźmy zbiór wierzchołków oznaczonych liczbami $1, \ldots, 4$ oraz permutację p:

$$p(1) = 2 \qquad p(2) = 4 \qquad p(3) = 3 \qquad p(4) = 1.$$

Istnieją tylko cztery turnieje, dla których ta permutacja jest automorfizmem:



Zadanie

Napisz program, który:

- *z pliku tekstowego* `AUT.IN` *wczyta opis permutacji n-elementowego zbioru wierzchołków,*
- *obliczy liczbę t różnych n-wierzchołkowych turniejów, dla których ta permutacja jest automorfizmem,*
- *zapisze w pliku tekstowym* `AUT.OUT` **resztę z dzielenia** t **przez** *1 000.*

Wejście

W pierwszym wierszu pliku tekstowego `AUT.IN` *znajduje się jedna liczba naturalna n, $1 \leqslant n \leqslant 10\,000$, będąca liczbą wierzchołków.*

W kolejnych n wierszach znajduje się opis permutacji p. Zakładamy, że wierzchołki są ponumerowane liczbami od 1 do n. W wierszu $(k+1)$-szym znajduje się wartość permutacji p dla wierzchołka nr k (tzn. wartość $p(k)$).

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `AUT.OUT` *powinna znaleźć się jedna liczba całkowita będąca resztą z dzielenia przez 1 000 liczby różnych n-wierzchołkowych turniejów, dla których permutacja p jest automorfizmem.*

PRZYKŁAD

Dla pliku wejściowego `AUT.IN`*:*

```
4
2
4
3
1
```

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

```
4
```

ROZWIĄZANIE

Rozważmy zbiór A wszystkich nieuporządkowanych par liczb ze zbioru $\{1, 2, \ldots, n\}$ — każdą parę liczb będziemy utożsamiać z parą wierzchołków o odpowiednich numerach. Pokażemy, że:

- albo nie istnieje żaden turniej, dla którego dana permutacja p jest automorfizmem,
- albo wyznacza ona podział zbioru A na rozłączne podzbiory $A_1, A_2, ..., A_s$, zwane dalej *gromadami*, dające w sumie zbiór A i takie, że:
  - zwrot krawędzi między dowolną parą wierzchołków z danej gromady determinuje zwrot krawędzi między pozostałymi parami wierzchołków z tej gromady,
  - zwroty krawędzi między parami wierzchołków należącymi do różnych gromad można wybrać niezależnie.

Szukana liczba t turniejów, dla których permutacja p jest automorfizmem będzie równa 2^s, ponieważ zwrot krawędzi pomiędzy parami wierzchołków z jednej gromady można ustalić na 2 sposoby. Pary należące do tej samej gromady nazwiemy parami *zależnymi.*

Cyklem permutacji p będziemy nazywać k-elementowy ciąg różnych liczb $(c_0, c_1, \ldots, c_{k-1})$ taki, że $c_i = p(c_{i-1})$, dla $1 \leqslant i < k$ i $c_0 = p(c_{k-1})$. Łatwo pokazać, że każdą permutację można przedstawić w postaci zbioru rozłącznych cykli.

Dla przykładu permutacja p:

$$p(1) = 2 \qquad p(2) = 5 \qquad p(3) = 6 \qquad p(4) = 1 \qquad p(5) = 4 \qquad p(6) = 3$$

składa się z dwóch cykli:

$$1 \longrightarrow 2 \longrightarrow 5 \longrightarrow 4 \qquad\qquad 3 \longrightarrow 6$$

Liczbę s gromad wyznaczonych przez daną permutację p można szybko znaleźć, znając rozbicie permutacji na cykle.

Weźmy cykl C o długości k oraz należące do niego elementy u i v. Połóżmy biały żeton na u i czerwony żeton na v. Jeśli oba żetony jednocześnie przesuniemy po cyklu o jeden element, to dostaniemy parę zależną od $\{u, v\}$. Aby wyznaczyć wszystkie takie pary powtarzamy ten ruch tak długo, aż oba żetony z powrotem znajdą się na pozycjach startowych u i v.

Zauważmy, że jeśli długość cyklu k jest liczbą parzystą i v jest odległe o $\frac{k}{2}$ od u na cyklu C, to po $k/2$ krokach czerwony żeton znajdzie się na u, a biały — na v. To oznacza sprzeczność — $u \to v$ jest krawędzią w turnieju wtedy i tylko wtedy, gdy $v \to u$ jest krawędzią w tym turnieju. W takim przypadku nie istnieje żaden turniej, dla którego permutacja p byłaby automorfizmem — ostateczną odpowiedzią jest liczba 0. Dalej będziemy zakładać, że wszystkie cykle są nieparzystej długości.

Jeśli długość cyklu jest liczbą nieparzystą, to niezależnie od tego jak wybraliśmy u i v, do początkowego ustawienia żetonów powrócimy po k krokach. Liczba k jest zatem licznością dowolnej gromady złożonej z par elementów cyklu C. Zbiór różnych par elementów cyklu C ma $k(k-1)/2$ elementów, zatem rozpada się na $(k-1)/2$ gromad.

Niech teraz u będzie elementem cyklu C o długości k, a v — elementem cyklu D długości l. Połóżmy znowu biały żeton na u, a czerwony na v. Aby wyznaczyć pary zależne od $\{u, v\}$, podobnie jak poprzednio synchronicznie przesuwamy żetony, tym razem po rozłącznych cyklach. Po ilu krokach wrócimy do sytuacji początkowej? Żeton biały wraca na u po k krokach, żeton czerwony wraca na v po l krokach, zatem musimy wykonać $NWW(k, l)$ kroków. Wszystkich par o jednym elemencie z C i drugim z D jest kl, a wszystkie gromady utworzone z takich par są liczności $NWW(k, l)$, więc mamy $kl/NWW(k, l) = NWD(k, l)$ takich gromad.

Możemy już policzyć liczbę gromad, na które rozpada się zbiór wszystkich par. Musimy najpierw wyznaczyć długości wszystkich cykli permutacji p — tę operację można łatwo zaimplementować, by działała w czasie $O(n)$ — a następnie posumować liczby gromad po wszystkich parach cykli, stosując wyżej wyprowadzone wzory. Ponieważ permutacja może mieć rzędu n cykli, nasz algorytm miałby złożoność czasową nie lepszą niż $O(n^2)$. Możemy jednak poprawić ten wynik poprzez zbiorowe traktowanie cykli o identycznych długościach. Niech c_k oznacza liczbę cykli o długości k. Wtedy sumaryczna liczba gromad utworzonych przez pary o elementach pochodzących

- z tego samego cyklu o długości k wynosi $\frac{c_k \cdot (k-1)}{2}$,
- z różnych cykli o długości k wynosi $\frac{c_k(c_k-1)}{2} \cdot k$,
- z cykli o różnych długościach k i l wynosi $c_k c_l \cdot NWD(k, l)$.

Ile może być różnych długości cykli? Aby było ich najwięcej, musimy wziąć po jednym cyklu o długości $1, 3, 5, \ldots$, przy czym suma długości cykli nie może być większa od n. Ponieważ $1+3+5+\ldots+\lceil 2\sqrt{n} \rceil \geqslant n$, różnych długości cykli jest co najwyżej $O(\sqrt{n})$. Do rozpatrzenia mamy zatem $O(n)$ par grup cykli różnych długości. Policzenie $NWD(p, q)$ algorytmem Euklidesa zajmuje czas $O(\log p + \log q)$ czasu (jego opis Czytelnik znajdzie w [13]), zatem otrzymujemy algorytm o złożoności czasowej $O(n \log n)$.

To nie koniec zadania, gdyż pozostaje nam jeszcze policzyć resztę z dzielenia $t = 2^s$ przez 1000. Kolejne potęgi dwójki będziemy naliczać iteracyjnie. Łatwo zauważyć, że przy wykonywaniu kolejnych mnożeń przez 2 wystarczy przechowywać jedynie resztę z dzielenia aktualnego wyniku przez 1000. Obserwacja ta jednak nie wystarcza, ponieważ s może być rzędu n^2. Zauważmy jednak, że po wystarczająco dużej liczbie kroków (ale na pewno mniejszej niż 1000) trafimy w końcu na wartość, którą otrzymaliśmy już wcześniej. Niech i będzie właśnie takim najmniejszym wykładnikiem, że $2^i \bmod 1000 = 2^{\mathrm{m}} \bmod 1000$ dla pewnego $m < i$. Wtedy dla każdego $j > i$ mamy

$$2^j \bmod 1000 = (2^i \bmod 1000) \cdot 2^{(j-i) \bmod (i-m)} \bmod 1000$$

Wystarczy zatem namnożyć dwójkę jeszcze $(j - i) \bmod (\mathrm{i} - \mathrm{m})$ razy. Aby zrealizować ten algorytm potrzebna nam będzie 1000-elementowa tablica, w której r-tym elementem będzie wykładnik i, dla którego otrzymaliśmy resztę r.

Algorytm ten wykona co najwyżej 2000 mnożeń, więc jego złożoność czasowa to $O(1)$ (jest stała)!

TESTY

Do sprawdzania rozwiązań zawodników użyto 12 testów. Testy 2 i 3 oraz 11 i 12 były zgrupowane. Permutacje z wszystkich testów, za wyjątkiem testu 2, zawierały wyłącznie cykle nieparzystych długości. Krótki opis testów zawarty jest poniżej:

- AUT1.IN — prosty test poprawnościowy,
- AUT2.IN — prosty test poprawnościowy z cyklem parzystej długości,
- AUT3.IN — prosty test do zgrupowania z 2,
- AUT4.IN — poprawnościowy test losowy, permutacja 50-elementowa o 4 cyklach,
- AUT5.IN — test poprawnościowy, permutacja 501-elementowa o jednym cyklu,

- AUT6.IN — wydajnościowy test losowy, permutacja 10000-elementowa o 100 cyklach,
- AUT7.IN — wydajnościowy test losowy, permutacja o 100 cyklach długości 1, 3, 5, ..., 199,
- AUT8.IN — wydajnościowy test losowy, permutacja 5000-elementowa o 4500 cyklach,
- AUT9.IN — wydajnościowy test losowy, permutacja 10000-elementowa o 5000 cyklach,
- AUT10.IN — wydajnościowy test losowy, permutacja 10000-elementowa o 8000 cyklach,
- AUT11.IN — 10000-elementowa permutacja identycznościowa,
- AUT12.IN — prosty test do zgrupowania z 11.

Wojciech Rytter
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Trójramienny dźwig

Trójramienny dźwig ustawia kontenery na wagonach kolejowych. Wagony są ponumerowane kolejno $1, 2, \ldots$. Na każdym wagonie można postawić co najwyżej jeden kontener. W jednym ruchu dźwig pobiera ze składowiska trzy kontenery i ustawia je na wagonach o numerach i, $i+p$ oraz $i+p+q$, albo na wagonach o numerach i, $i+q$ oraz $i+p+q$ (dla pewnych stałych $p, q \geqslant 1$). Dźwig trzeba zaprogramować tak, żeby załadował kontenerami pierwsze n wagonów pociągu (pociąg ma $n+p+q$ wagonów). Program składa się z ciągu instrukcji. Każda z instrukcji opisuje jeden ruch dźwigu. Instrukcja programu ma postać trójki (x, y, z), gdzie $1 \leqslant x < y < z \leqslant n+p+q$, i określa numery wagonów, na które dźwig ma ustawić kontenery. Jeżeli po wykonaniu programu na każdym spośród n pierwszych wagonów pociągu znajduje się dokładnie jeden kontener, to mówimy, że program jest **poprawny**.

Zadanie

Napisz program, który:

- *z pliku tekstowego* `TRO.IN` *wczyta charakterystykę dźwigu (liczby p i q) oraz liczbę wagonów do załadowania (n),*
- *wygeneruje poprawny program dla dźwigu,*
- *zapisze go do pliku tekstowego* `TRO.OUT`.

Wejście

W pierwszym i jedynym wierszu pliku tekstowego `TRO.IN` *znajdują się dokładnie trzy dodatnie liczby całkowite pooddzielane pojedynczymi odstępami. Są to odpowiednio: liczby p i q określające parametry dźwigu oraz liczba n, będąca liczbą początkowych wagonów pociągu do załadowania. Liczby te spełniają nierówności $1 \leqslant n \leqslant 300\,000$, $2 \leqslant p+q \leqslant 60\,000$.*

Wyjście

W pierwszym wierszu pliku tekstowego `TRO.OUT` *powinna znajdować się dokładnie jedna liczba całkowita m będąca liczbą instrukcji w wygenerowanym programie. W każdym z kolejnych m wierszy powinny znajdować się dokładnie trzy liczby naturalne x, y, z pooddzielane pojedynczymi odstępami, $1 \leqslant x < y < z \leqslant n+p+q$, $x \leqslant n$, $y \in \{x+p, x+q\}$, $z = x+p+q$. Są to numery wagonów, na które dźwig ma położyć kontenery w kolejnym ruchu.*

Przykład

Dla pliku wejściowego `TRO.IN`:

```
2 3 10
```

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

```
4
1 3 6
2 4 7
5 8 10
9 11 14
```

ROZWIĄZANIE

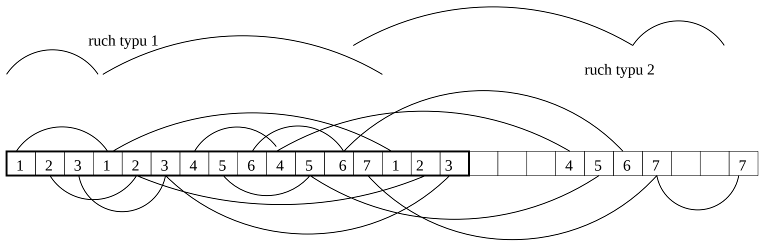
Przyjmijmy, że $p \leqslant q$. Ruch typu $i, i+p, i+p+q$ nazwijmy ruchem typu 1, a $i, i+q, i+p+q$ ruchem typu 2, gdzie i jest najmniejszym numerem wolnego wagonu.

Opis algorytmu

Rozwiązanie polega na wykonywaniu tak długo jak to możliwe, ruchów typu 1, a gdy to nie jest możliwe ruchu typu 2. Okazuje się, że takie rozwiązanie zawsze generuje prawidłowy program.

Złożoność czasowa takiego algorytmu wynosi $O(n+p+q)$, pamięciowa — $O(p+q)$.

Rys. 1 Przykładowa historia działania dźwigu trójramiennego. W każde pole wpisany jest kolejny numer ruchu, w którym to pole zostaje zajęte. Jedynie 7-my ruch jest typu 2.



Na przykład, jeśli charakterystyką dźwigu jest $(3, 10)$ oraz $n = 16$, to jeden z możliwych programów dźwigu jest przedstawiony na rysunku 1.

Dowód poprawności

Załóżmy, że dla pewnego wagonu $i = s$ po raz pierwszy nie można wykonać żadnego z ruchów. Łatwo zauważyć, że wagon $i+p+q$ jest wolny. Również s jest wolny, natomiast $s+p$, $s+q$ są zajęte, ponieważ nie można wykonać ruchu. Wagon $s+q$ został zajęty w czasie wykonywania ruchu typu 2 dla pozycji $j = s-p$. Ale z tego wynika, że w momencie wykonywania ruchu dla $i = j$ wolne były $i, j+p, j+p+q$, zatem można było wykonać ruch typu 1, który ma priorytet i zająć pozycje $j, s, s+q$. Pozycja s nie może być zatem wolna — sprzeczność.

Obserwacja

Można się zastanowić nad podobnym problem dla dźwigu czteroramiennego. Jest to trudne zadanie, ponieważ najpierw dla danego n trzeba sprawdzić, czy w ogóle istnieje odpowiedni program dźwigu — nie każdy ciąg kolejnych n liczb da się wtedy "pokryć". Na przykład nie ma pokrycia dla $n = 3$ i dźwigu o charakterystyce (1, 2, 1), tzn. takiego który zapełnia wagony $i, i+1, i+3, i+4$.

TESTY

Do sprawdzania rozwiązań zawodników użyto 24 testów, których charakterystyki znajdują się poniżej.

Test	p	q	n
0	2	3	10
1	2	3	5
2	2	3	12
3	2	3	20
4	5	2	10
5	2	5	20
6	2	5	30
7	10	20	30
8	20	10	5
9	20	10	30
10	20	20	20
11	20	20	100
12	50	67	100
13	68	37	200
14	123	1234	1000
15	1234	123	2000
16	1000	2000	10000
17	2000	1000	20000
18	10000	10001	20000
19	10001	10000	30000
20	17777	20000	50000
21	20000	17777	60000
22	25000	35000	200000
23	35000	25000	300000

Następujące testy zostały zgrupowane: (1,2 i 3), (4,5 i 6), (7,8 i 9), (10 i 11), (12 i 13), (14 i 15), (16 i 17), (18 i 19), (20 i 21), (22 i 23).

Wojciech Guzicki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Kod

Drzewo binarne *może byc puste, albo składać sie z wierzchołka, do którego przyczepione są dwa drzewa, tzw. lewe i prawe poddrzewo. W każdym wierzchołku zapisana jest jedna litera alfabetu angielskiego. Wierzchołek drzewa, który nie znajduje się w żadnym poddrzewie, nazywamy* **korzeniem**. *Mówimy, że drzewo jest* **binarnym drzewem poszukiwań (BST)***, jeżeli dla każdego wierzchołka spełniony jest warunek, mówiący, że wszystkie litery z lewego poddrzewa wierzchołka występują w alfabecie wcześniej, niż litera zapisana w wierzchołku, natomiast wszystkie litery z prawego poddrzewa — później.* **Kodem drzewa BST** *nazywamy:*

- *ciąg pusty (0-elementowy), gdy drzewo jest puste,*
- *ciąg liter zaczynający się od litery zapisanej w korzeniu drzewa, po którym następuje kod lewego poddrzewa, a następnie kod prawego poddrzewa.*

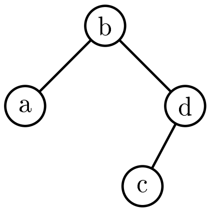
Rozważmy wszystkie k-wierzchołkowe drzewa BST, w wierzchołkach których umieszczono początkowe k liter alfabetu angielskiego. Wyobraźmy sobie listę kodów tych drzew, wypisanych w kolejności alfabetycznej. **(n,k)-kodem** *nazywamy n-ty kod na tej liście.*

Przykład

Istnieje dokładnie 14 kodów 4-wierzchołkowych binarnych drzew poszukiwań, konkretnie (w kolejności alfabetycznej):

`abcd abdc acbd adbc adcb bacd badc cabd cbad dabc dacb dbac dcab dcba`

Napis `badc` *jest (7,4)-kodem i odpowiada mu następujące drzewo BST:*



Zadanie

Napisz program, który:

- *z pliku tekstowego* `KOD.IN` *wczyta liczby n i k,*
- *wyznaczy (n,k)-kod,*
- *zapisze go do pliku tekstowego* `KOD.OUT`.

Wejście

W pierwszym i jedynym wierszu pliku tekstowego `KOD.IN` *zapisane są dokładnie dwie dodatnie liczby całkowite n i k, oddzielone pojedynczym znakiem odstępu, $1 \leqslant k \leqslant 19$. Liczba n nie przekracza liczby kodów drzew BST o k wierzchołkach.*

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `KOD.OUT` *powinno znajdować się słowo złożone z małych liter alfabetu angielskiego będące (n,k)-kodem.*

PRZYKŁAD

Dla pliku wejściowego `KOD.IN`*:*
```
11 4
```
poprawną odpowiedzią jest plik tekstowy `KOD.OUT`*:*
```
dacb
```

ROZWIĄZANIE

Zastanówmy się najpierw, ile jest drzew binarnych o zadanej liczbie wierzchołków. Oznaczmy przez C_k liczbę drzew mających k wierzchołków. Oczywiście $C_0 = 1$: tylko drzewo puste ma zero wierzchołków. Również jest tylko jedno drzewo mające jeden wierzchołek, zatem $C_1 = 1$. Niech teraz k będzie liczbą naturalną większą od jedności. Drzewo mające k wierzchołków składa się z korzenia, do którego przyczepione są dwa poddrzewa mające łącznie $k-1$ wierzchołków. Mamy więc k możliwości:

- Przypadek 0. Lewe poddrzewo ma 0 wierzchołków, prawe ma $k-1$ wierzchołków (takich drzew jest oczywiście $C_0 \cdot C_{k-1}$).
- Przypadek 1. Lewe poddrzewo ma 1 wierzchołek, prawe ma ich $k-2$ (takich drzew jest $C_1 \cdot C_{k-2}$);

  ...
- Przypadek $k-1$. Lewe poddrzewo ma $k-1$ wierzchołków, prawe ma 0 wierzchołków (takich drzew jest $C_{k-1} \cdot C_0$).

Stąd otrzymujemy następujący wzór rekurencyjny na liczbę drzew:

$$C_k = C_0 \cdot C_{k-1} + C_1 \cdot C_{k-2} + \ldots + C_{k-1} \cdot C_0 = \sum_{i=0}^{k-1} C_i \cdot C_{k-i-1}.$$

Liczby C_k nazywamy liczbami Catalana. Występują one w wielu innych zadaniach kombinatorycznych. Na przykład:

- Liczba funkcji niemalejących
  $$f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\},$$
  tzn. takich, że jeśli $k \leqslant l$, to $f(k) \leqslant f(l)$ dla dowolnych $k, l \in \{1, 2, \ldots, n\}$ i spełniających warunek $f(k) \leqslant k$, dla każdego $k \in \{1, 2, \ldots, n\}$, jest równa C_n.
- Liczba różnych triangulacji (czyli sposobów podziału przekątnymi na trójkąty) wielokąta wypukłego mającego n boków jest równa C_{n-2}.
- Przypuśćmy, że mamy dane działanie dwuargumentowe $\circ$ i rozważamy wyrażenie postaci $x_1 \circ x_2 \circ \ldots \circ x_n$. W tym wyrażeniu możemy rozstawić nawiasy (tak, by zawsze wykonywać działanie na dwóch argumentach) na wiele sposobów. Na przykład, dla $n = 4$ mamy następujące sposoby:
  $$(x_1 \circ x_2) \circ (x_3 \circ x_4), \quad ((x_1 \circ x_2) \circ x_3) \circ x_4, \quad x_1 \circ (x_2 \circ (x_3 \circ x_4)),$$
  $$(x_1 \circ (x_2 \circ x_3)) \circ x_4, \quad x_1 \circ ((x_2 \circ x_3) \circ x_4.$$
  Liczba różnych sposobów rozstawienia nawiasów jest równa C_{n-1}. Inaczej mówiąc, jeśli działanie $\circ$ jest niełączne (przykładem takiego działania jest dzielenie liczb rzeczywistych dodatnich), to wykonując to działanie na wszystkie możliwe sposoby, z zachowaniem kolejności $x_1, x_2, \ldots, x_n$, możemy otrzymać co najwyżej C_{n-1} różnych wyników.

- Liczby Catalana występują również jako rozwiązania następującego zadania: w kolejce do kina stoi $2n$ osób, n z nich ma tylko 5 złotych, każda z pozostałych ma tylko banknot dziesięciozłotowy. Bilet kosztuje 5 złotych. W kasie nie ma pieniędzy, więc może się okazać, że w którymś momencie kasjerka nie będzie mogła wydać reszty kolejnej osobie (na przykład, jeśli pierwsza osoba ma 5 złotych, a dwie następne po 10 złotych, to trzecia osoba nie otrzyma reszty i kolejka się zatrzyma). Okazuje się, że liczba sposobów ustawienia osób w kolejce tak, by wszyscy mogli kupić bilety, jest równa C_n.

Czytelnikowi zainteresowanemu kombinatoryką polecamy książki *W. Lipski, W. Marek – Analiza kombinatoryczna* i *V. Bryant – Aspekty kombinatoryki*, z których można dowiedzieć się więcej o liczbach Catalana.

Podany wyżej wzór rekurencyjny pozwala obliczyć kolejne liczby Catalana. Początkowe liczby Catalana są równe odpowiednio:

$$\begin{aligned} C_0 &= 1,\\ C_1 &= 1,\\ C_2 &= 2,\\ C_3 &= 5,\\ C_4 &= 14,\\ C_5 &= 42,\\ C_6 &= 132,\\ C_7 &= 429,\\ C_8 &= 1430,\\ C_9 &= 4862,\\ C_{10} &= 16796. \end{aligned}$$

Znany jest też wzór ogólny na k-tą liczbę Catalana. Mianowicie

$$C_k = \frac{1}{k+1} \cdot \binom{2k}{k}.$$

W programie wzorcowym liczby Catalana są obliczane za pomocą wzoru rekurencyjnego.

Teraz opiszemy algorytm znajdowania (n, k)-kodu drzewa binarnego. Najpierw wyjaśnimy istotę działania tego algorytmu na przykładzie. Przyjmijmy $k = 10$ i $n = 8751$. Wiemy już, że istnieje 16796 drzew binarnych mających 10 wierzchołków. Ta liczba jest sumą następujących 10 liczb:

1. liczby $C_0 \cdot C_9$, równej 4862; jest to liczba tych drzew, w których lewe poddrzewo ma 0 wierzchołków, a prawe 9 wierzchołków; są to dokładnie te drzewa binarne, które mają w korzeniu umieszczoną literę `a`;
2. liczby $C_1 \cdot C_8$, równej 1430; jest to liczba tych drzew, w których lewe poddrzewo ma 1 wierzchołek, a prawe 8 wierzchołków; są to drzewa, które mają w korzeniu umieszczoną literę `b`;
3. liczby $C_2 \cdot C_7$, równej 858; jest to liczba tych drzew, w których lewe poddrzewo ma 2 wierzchołki, a prawe 7 wierzchołków; są to drzewa, które mają w korzeniu umieszczoną literę `c`;

   ...

10. liczby $C_9 \cdot C_0$, równej 4862; jest to liczba tych drzew, w których lewe poddrzewo ma 9 wierzchołki, a prawe 0 wierzchołków; są to drzewa, które mają w korzeniu umieszczoną literę `j`.

Kody wszystkich drzew z grupy pierwszej poprzedzają oczywiście w porządku alfabetycznym kody drzew z grupy drugiej, te z kolei poprzedzają kody drzew w grupy trzeciej itd. Zauważmy teraz, że łączna liczba kodów zaczynających się od liter `a`, `b`, `c`, `d`, `e` jest równa 8398 (czyli mniej niż 8751), a jeśli dopuścimy jeszcze literę `f`, to liczba kodów wyniesie 8986 (a więc więcej niż 8751). Stąd wynika, że pierwszą literą kodu jest `f`. Po niej nastąpią litery od `a` do `e` (w nieznanej jeszcze kolejności) i na końcu litery od `g` do `j` (również w nieznanej kolejności). W ten sposób ustaliliśmy pierwszą literę kodu drzewa. Następnie będziemy chcieli ustalić kody obu poddrzew. W tym celu obliczymy kolejne numery tych kodów i rekurencyjnie znajdziemy same kody.

Każdemu z C_5 kodów złożonych z liter od `a` do `e` na początku odpowiada C_4 kodów złożonych z liter od `g` do `j` na końcu. Mamy znaleźć 353-ci z kolei kod rozpoczynający się literą `f` (gdyż $8751 - 8398 = 353$). Każdemu kodowi lewego poddrzewa odpowiada 14 kodów prawych poddrzew. Ale $352 = 25 \cdot 14 + 2$. Stąd wynika, że odcinek początkowy naszego kodu (po literze `f`) ma być dwudziestym szóstym z kolei kodem składającym się z liter od `a` do `e`, a odcinek końcowy ma być trzecim z kolei kodem składającym się z liter od `g` do `j`. Jest tak dlatego, że pierwszym dwudziestu pięciu kodom lewych poddrzew odpowiada tylko $25 \cdot 14$ czyli 350 kodów całych drzew. Nasz kod ma więc być trzecim kodem z następnej, tzn. dwudziestej szóstej grupy. Powstaje pytanie, dlaczego dzielimy 352, a nie 353. Mianowicie mamy proste wzory ogólne na numer grupy (czyli numer kodu lewego poddrzewa) i numer w tej grupie (czyli numer kodu prawego poddrzewa):

> przypuśćmy, że nasz kod ma być m-tym kodem zaczynającym się od pewnej znanej już litery c (w naszym przypadku $c =$ `f`); znamy więc wielkości obu poddrzew: niech lewe składa się z l wierzchołków, a prawe z p wierzchołków. Wtedy lewe poddrzewo ma numer $(m-1)$ **div** $C_p + 1$, a prawe $(m-1)$ **mod** $C_p + 1$.

Kontynuując obliczenia w naszym przykładzie zauważymy, że odcinek początkowy kodu zaczyna się literą `d`, po której następuje trzeci kod składający się z liter `a`, `b`, `c` i jedyny kod składający się z litery `e`. Odcinek końcowy zaczyna się literą `g`, po której następuje trzeci kod składający się z liter `h`, `i`, `j`. Ostatecznie otrzymamy kod `fdbacegihj`.

Teraz już nietrudno napisać odpowiedni algorytm rekurencyjny. Główna funkcja wyznaczająca (n,k)-kod ma w nim postać następującą. Argumentami funkcji są: litera *cp*, od której zaczynają się litery umieszczone w wierzchołkach drzewa (będzie to potrzebne w przypadku prawego poddrzewa, w którym występują litery nie od początku alfabetu), numer n drzewa i liczba wierzchołków k. W tablicy *Catalan* umieszczone są oczywiście liczby Catalana, obliczone wcześniej za pomocą wzoru rekurencyjnego. Na początku funkcja zwraca wartości w przypadkach najprostszych: kod pusty dla drzewa pustego i kod jednoliterowy *cp* dla drzewa składającego się z jednego wierzchołka. Dla drzew większych obliczamy w pętli sumę iloczynów liczb Catalana dotąd, aż przekroczy ona n. Zauważmy, że wartości zmiennych l i p są wtedy równe dokładnie liczbom wierzchołków w lewym i prawym poddrzewie. Korzeń otrzymujemy znajdując literę stojącą w alfabecie o l miejsc dalej od *cp*. Potem zgodnie z powyższymi wzorami znajdujemy numery lewego i prawego poddrzewa i dwukrotnie wywołujemy procedurę rekurencyjnie. Otrzymane kody drzew łączymy wreszcie ze znalezionym wcześniej korzeniem. Oto treść tej procedury:

```
function Kod (cp : char; n : longint; k : integer) : String;
  { cp : litera początkowa kodu drzewa;
     n  : numer kodu drzewa wśród drzew mających wierzchołki
          oznaczone literami począwszy od cp zakładamy, że liczba
          n nie przekracza liczby Catalana C[k];
     k  : liczba wierzchołków drzewa}
var
  l,p : integer; {liczba wierzchołków w lewym i prawym poddrzewie}
  Suma : longint; {suma iloczynów liczb Catalana}
  SumaPoprzednia : longint;
  m : longint;
  n1,n2 : longint; {numery lewego i prawego poddrzewa}
  znak : char;
begin
  if (k=0) then
    Kod:=''
```

```
    else if (k=1) then
      Kod:=cp
    else begin
      Suma:=0;
      l:=–1;
      p:=k;
      repeat
        l:=l+1;
        p:=p-1;
        SumaPoprzednia:=Suma;
        Suma:=Suma+Catalan[l]*Catalan[p];
      until (Suma≥n);
      znak:=chr(ord(cp)+1);
      m:=n-SumaPoprzednia;
      n1:=((m-1) div Catalan[p])+1;
      n2:=((m-1) mod Catalan[p])+1;
      Kod:=znak+Kod(cp,n1,l)+Kod(succ(znak),n2,p)
    end
end;
```

Ten najprostszy program zamieszczony jest w pliku o nazwie `kod0.pas`. Nie jest on jednak optymalny. Można zauważyć, że pewne obliczenia są wykonywane wielokrotnie, na przykład sumowanie iloczynów liczb Catalana. Te iloczyny i ich sumy można obliczyć wcześniej i umieścić w tablicy. Tak napisany program znajduje się w pliku `kod.pas`.

Ograniczenie $k = 19$ wynika z wielkości liczb Catalana. Liczba $C_{19} = 1767263190$ mieści się jeszcze w zakresie liczb typu *longint*.

TESTY

Do sprawdzania rozwiązań zawodników użyto 17 testów:

- KOD0.IN — test z treści zadania,
- KOD1.IN — $n = 3$, $k = 3$,
- KOD2.IN — $n = 42$, $k = 5$,
- KOD3.IN — $n = 1$, $k = 6$,
- KOD4.IN — $n = 2000$, $k = 9$,
- KOD5.IN — $n = 50000$, $k = 11$,
- KOD6.IN — $n = 200000$, $k = 13$,
- KOD7.IN — $n = 1000000$, $k = 15$,
- KOD8.IN — $n = 20000000$, $k = 17$,
- KOD9.IN — $n = 100000000$, $k = 18$,
- KOD10.IN — $n = 400000000$, $k = 19$,
- KOD11.IN — $n = 800000000$, $k = 19$,
- KOD12.IN — $n = 100$, $k = 19$,
- KOD13.IN — $n = 1767263190$, $k = 19$,
- KOD14.IN — $n = 1111111111$, $k = 19$,

- KOD15.IN — $n = 1234567890$, $k = 19$,
- KOD16.IN — $n = 608197699$, $k = 19$.

Marcin Kubica
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Labirynt studni

Wewnątrz Bajtogóry znajduje się mityczny labirynt studni. Wejście do labiryntu znajduje się na szczycie góry. Labirynt składa się z wielu komnat. Każda z nich jest w jednym z trzech kolorów: czerwonym, zielonym lub niebieskim. Dwie komnaty tego samego koloru wyglądają identycznie i są nieodróżnialne. W każdej komnacie znajdują się trzy studnie oznaczone numerami 1, 2 i 3. Między komnatami można poruszać się tylko w jeden sposób — wskakując do jednej ze studni spada się (niekoniecznie pionowo) do jednej z komnat położonych niżej. Z komnaty, w której znajduje się wejście do labiryntu można przedostać się w ten sposób do każdej z pozostałych komnat. Wszystkie drogi przez labirynt prowadzą do smoczej jamy znajdującej się na samym jego dnie. Każdemu przejściu przez labirynt odpowiada ciąg numerów studni wybieranych w kolejno odwiedzanych komnatach. Ciąg ten nazywa się **planem podróży**.

W smoczej jamie żyje Bajtosmok. Legenda głosi, że ten, kto przedstawi Bajtosmokowi dokładny plan labiryntu, otrzyma wielkie skarby. Wszystkich innych Bajtosmok potężnym kopnięciem wyprasza z wnętrza góry.

Śmiałek o imieniu Bajtazar wielokrotnie przemierzał labirynt i opracował jego mapę. Jednak Bajtosmok orzekł, że co prawda wszystkie komnaty znajdują się na mapie, ale wiele komnat jest na niej powtórzonych.

— Ja też — powiedział poklepując Bajtazara po ramieniu — projektując labirynt narysowałem podobny rysunek, ale szybko stwierdziłem, że komnat można zrobić o wiele mniej, a gość poruszający się według dowolnego planu podróży i tak będzie oglądał takie same sekwencje komnat. Pomyślałem trochę i maksymalnie zredukowałem projekt.

Zadanie

Napisz program, który:

- *wczyta mapę Bajtazara z pliku tekstowego* `LAB.IN`,
- *obliczy prawdziwą liczbę komnat w labiryncie,*
- *zapisze wynik do pliku tekstowego* `LAB.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `LAB.IN` *zapisana jest jedna liczba całkowita n, $2 \leqslant n \leqslant 6\,000$, będąca liczbą komnat (łącznie ze smoczą jamą). Komnaty są ponumerowane od 1 do n tak, że komnaty o większych numerach znajdują się niżej (komnata, w której znajduje się wejście do labiryntu ma numer 1, a smocza jama ma numer n). W kolejnych $n-1$ wierszach pliku są opisane komnaty (poza smoczą jamą) oraz studnie prowadzące od nich w dół. W każdym z tych wierszy znajduje się litera, po niej jeden znak odstępu, a następnie trzy liczby całkowite oddzielone pojedynczymi odstępami. Litera oznacza kolor komnaty (*`C` *— czerwony,* `Z` *— zielony,* `N` *— niebieski), a i-ta liczba (dla $i = 1, 2, 3$) jest numerem komnaty, do której prowadzi i-ta studnia.*

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `LAB.OUT` *powinna znaleźć się dokładnie jedna liczba całkowita będąca minimalną liczbą komnat w labiryncie (łącznie ze smoczą jamą), przy której podróżnik poruszający się według dowolnego planu obserwuje taką samą sekwencję komnat, jak dla labiryntu opisanego w pliku wejściowym.*

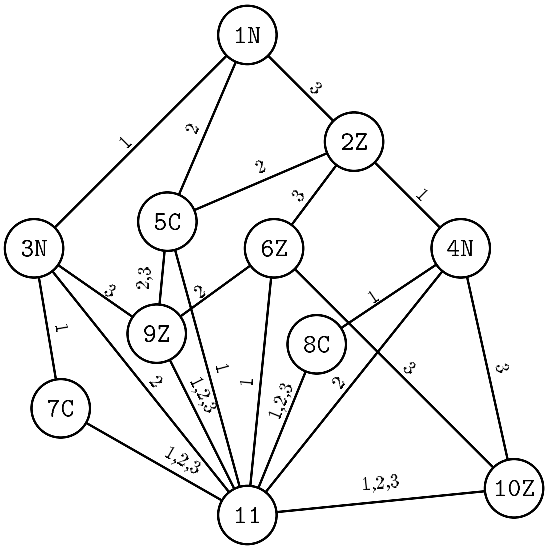
Przykład

Dla pliku wejściowego `LAB.IN`:

```
11
N 3 5 2
Z 4 5 6
N 7 11 9
N 8 11 10
```

```
C 11 9 9
Z 11 9 10
C 11 11 11
C 11 11 11
Z 11 11 11
Z 11 11 11
```

opisującego następujący labirynt:



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

```
8
```

ROZWIĄZANIE

Sformułujmy to zadanie w terminologii grafowej. Plan labiryntu to multigraf* skierowany bez cykli — jego wierzchołki reprezentują komnaty, a krawędzie reprezentują studnie. Każdy wierzchołek (z wyjątkiem smoczej jamy) jest pokolorowany jednym z trzech kolorów: czerwonym, zielonym lub niebieskim. Z każdego wierzchołka wychodzą po trzy krawędzie etykietowane liczbami 1, 2 i 3, z wyjątkiem smoczej jamy, z której nie wychodzą żadne krawędzie. Dla każdego wierzchołka na planie istnieje ścieżka prowadząca do niego z wierzchołka reprezentującego wejście do labiryntu. Podobnie, dla każdego wierzchołka istnieje ścieżka prowadząca z niego do smoczej jamy.

Każdej ścieżce prowadzącej z wejścia do labiryntu do smoczej jamy odpowiada plan podróży, będący ciągiem etykiet krawędzi tworzących daną ścieżkę, oraz ciąg kolorów wierzchołków tworzących tę ścieżkę. Równocześnie plan podróży wyznacza jednoznacznie odpowiadającą mu ścieżkę łączącą wejście do labiryntu ze smoczą jamą. Możemy powiedzieć, że dwa plany labiryntów są sobie równoważne, jeżeli:

- mają one takie same zbiory możliwych planów podróży, oraz
- dla każdego z tych planów podróży, odpowiadające im w obu multigrafach ścieżki łączące wejścia do labiryntów ze smoczymi jamami, charakteryzują się takimi samymi sekwencjami kolorów wierzchołków.

Zadanie polega na znalezieniu planu labiryntu o najmniejszej liczbie wierzchołków, równoważnego zadanemu planowi. Zastanówmy się, jak mógłby wyglądać taki minimalny plan.

Przez głębokość wierzchołka oznaczamy długość najdłuższej ścieżki prowadzącej z tego wierzchołka do smoczej jamy. Przyjmujemy, że smocza jama ma głębokość 0. Zbiór wszystkich wierzchołków o danej głębokości nazywamy warstwą.

* Multigraf różni się tym od grafu, że może mieć wiele krawędzi łączących tę samą parę wierzchołków.

Powiedzieliśmy, że każdej ścieżce łączącej wejście do labiryntu ze smoczą jamą odpowiada pewien plan podróży i jednocześnie ten plan podróży jednoznacznie wyznacza tę ścieżkę. Fakt ten można uogólnić. Każdej ścieżce możemy przyporządkować jej "plan" będący ciągiem etykiet krawędzi tworzących tę ścieżkę. Równocześnie taki plan, wraz z początkowym wierzchołkiem, wyznacza jednoznacznie tę ścieżkę.

Charakterystyką ścieżki będziemy nazywać parę złożoną z planu ścieżki oraz ciągu kolorów wierzchołków tworzących tę ścieżkę. Dla każdego wierzchołka definiujemy jego charakterystykę, jako zbiór charakterystyk wszystkich ścieżek prowadzących z tego wierzchołka do smoczej jamy.

Zauważmy, że wierzchołki mające taką samą charakterystykę należą do tej samej warstwy. Ponadto, jeśli z wierzchołków o takiej samej charakterystyce przejdziemy równocześnie wzdłuż krawędzi o takiej samej etykiecie, to dojdziemy do wierzchołków o takiej samej charakterystyce. Tak więc, w szukanym planie nie może być dwóch wierzchołków o takiej samej charakterystyce — gdyby były, to moglibyśmy skleić ze sobą dwa wierzchołki o takiej samej charakterystyce i minimalnej głębokości, uzyskując plan labiryntu równoważny danemu, ale o mniejszej liczbie wierzchołków. Ponadto, dla dowolnego wierzchołka v w danym planie labiryntu i dla dowolnej ścieżki prowadzącej od wejścia do labiryntu do v, w szukanym planie istnieje ścieżka o takiej samej charakterystyce, prowadząca od wejścia do labiryntu do wierzchołka o takiej samej charakterystyce, jak charakterystyka v. Wynika stąd, że szukany plan labiryntu możemy uzyskać sklejając wierzchołki o takich samych charakterystykach.

Sklejając wierzchołki nie musimy wyznaczać ich charakterystyk. Możemy skorzystać z następującego prostego faktu. Niech v i w będą dwoma wierzchołkami o głębokości h. Jeżeli wszystkie wierzchołki o głębokościach mniejszych od h mają różne charakterystyki, to v i w mają takie same charakterystyki wtedy i tylko wtedy, gdy krawędzie o takich samych etykietach, wychodzące z v i w prowadzą do takich samych wierzchołków. Fakt ten wynika bezpośrednio z definicji charakterystyki. Tak więc, najlepiej sklejać wierzchołki o tych samych charakterystykach z kolejnych warstw o rosnących głębokościach.

Nasze rozwiązanie składa się z dwóch faz: podzielenia wierzchołków na warstwy i sklejania wierzchołków z kolejnych warstw. Jak jednak podzielić wierzchołki na warstwy? Możemy posłużyć się tu algorytmem podobnym do sortowania topologicznego. Warstwy wyznaczamy w kolejności rosnących głębokości. W momencie, gdy wstawiamy jakiś wierzchołek do warstwy, to "przeglądamy" wszystkie krawędzie wchodzące do tego wierzchołka. Jeżeli wszystkie trzy krawędzie wychodzące z któregokolwiek wierzchołka zostaną przejrzane, to możemy wstawić ten wierzchołek do warstwy o głębokości o jeden większej od głębokości wierzchołka, do którego prowadzi ostatnia przejrzana krawędź.

Z każdym wierzchołkiem możemy mieć związany licznik, który początkowo jest równy 3. Wówczas "przejrzenie" krawędzi polega na zmniejszeniu o 1 licznika związanego z wierzchołkiem, z którego dana krawędź wychodzi. Gdy licznik osiągnie 0, to wszystkie krawędzie wychodzące z danego wierzchołka zostały przejrzane.

Do zaimplementowania opisanego algorytmu może być przydatna następująca struktura danych. Dla każdego wierzchołka pamiętamy jego kolor, krawędzie wychodzące z tego wierzchołka, krawędzie wchodzące do tego wierzchołka oraz pewne dodatkowe informacje potrzebne w trakcie sklejania wierzchołków, które omówimy dalej.

```
const MAXN = 6000; { Maksymalna liczba wierzchołków. }
type
  PElem = ↑TElem; { Listy nieujemnych liczb całkowitych. }
  TElem = record
    l: Word;
    nast: PElem
  end
  TKolejka = record { Kolejki nieujemnych liczb całkowitych. }
    pocz, kon: PElem
  end
  TListy = array [0..MAXN] of PElem; { Tablica list. }
  TKolejki = array [1..MAXN] of TKolejka; { Tablica kolejek. }
var
  n: Word; { Liczba wierzchołków. }
  V: array [1..MAXN] of record { Wierzchołki. }
    kolor: Byte; { Kolor. }
```

```
      numer: Word; { Numeracja używana przy sklejaniu wierzchołków. }
      dol: array [1..3] of Word { Krawędzie wychodzące z wierzchołków. }
    end
    Gora: ↑TListy; { Krawędzie prowadzące do wierzchołków. }
    Warstwy: ↑TListy; { Warstwy o kolejnych głębokościach. }
```

Zwróćmy uwagę, że ze względu na ograniczoną wielkość pojedynczej tablicy, dane o wierzchołkach są pamiętane w trzech tablicach.

Funkcja dzieląca wierzchołki na warstwy wygląda następująco:

```
procedure Wstaw(var L: PElem; i: Word);
{ Wstawia liczbę i na początek listy L. }
  ⋮
function WyznaczWarstwy: Word;
{ Wyznacza warstwy, w wyniku daje liczbę warstw. }
var
  i, gleb: Word;
  P, Q: PElem;
  T: array [0..MAXN] of Byte; { Liczby nieodwiedzonych krawędzi. }
begin
  for i := 0 to n do begin
    Warstwy↑[i] := nil;
    T[i] := 3
  end;
  Wstaw(Warstwy↑[0], n);
  gleb := 0; { Głębokość aktualnie przetwarzanej warstwy. }
  while Warstwy↑[gleb] ≠ nil do begin
    { Wyznacz kolejne warstwy. }
    P := Warstwy↑[gleb];
    while P ≠ nil do begin
      { Przejrzyj wszystkie wierzchołki warstwy. }
      i := P↑.l;
      Q := Gora↑[i];
      while Q ≠ nil do begin
        { Przejrzyj wszystkie krawędzie wchodzące do danego }
        { wierzchołka. }
        Dec(T[Q↑.l]);
        if T[Q↑.l] = 0 then
          Wstaw(Warstwy↑[gleb+1], Q↑.l);
          { Wierzchołek, z którego prowadzi krawędź, }
          { należy do następnej warstwy. }
        Q := Q↑.nast
      end;
      P := P↑.nast
    end;
    Inc(gleb)
  end;
  WyznaczWarstwy := gleb-1
end;
```

Każdy wierzchołek jest raz wstawiany do swojej warstwy i raz przeglądany. Każda krawędź jest przeglądana dokładnie raz. Tak więc złożoność czasowa pierwszej fazy algorytmu jest rzędu n, gdzie n jest liczbą wierzchołków.

Druga faza algorytmu polega na sklejaniu w kolejnych warstwach wierzchołków o takich samych charakterystykach. Każdemu wierzchołkowi możemy przyporządkować "kod" postaci $\langle k, v_1, v_2, v_3 \rangle$, gdzie

k to kolor wierzchołka, a v_1, v_2, v_3, to wierzchołki, do których prowadzą krawędzie wychodzące z danego wierzchołka. Przetwarzając wierzchołki z kolejnej warstwy sklejamy wierzchołki o takich samych kodach. Jak jednak wyznaczyć, które wierzchołki mają takie same kody? Możemy skorzystać ze struktury słownikowej (dowolnego rodzaju, o dostępie w czasie logarytmicznym) — w momencie, gdy wstawiamy do struktury kod, który już w nim jest, to odpowiednie wierzchołki należy skleić. W ten sposób wyznaczenie wierzchołków do sklejenia wymaga czasu $O(n \log n)$, gdzie n jest liczbą wierzchołków. Jeżeli zamiast słownika o dostępie w czasie logarytmicznym użyjemy tablicy haszującej, to możemy uzyskać *oczekiwany* czas $O(n)$.

Można jednak wyznaczyć wierzchołki do sklejenia w *pesymistycznym* czasie $\Theta(n)$. Oznaczmy przez h głębokość wierzchołka z wejściem do labiryntu, przez n_i oznaczmy liczbę wierzchołków w warstwie głębokości i. Zauważmy, że $n_0 = n_h = 1$ oraz $\Sigma_{i=0}^{h} n_i = n$. W i-tym kroku drugiej fazy algorytmu sklejamy w i-tej warstwie wierzchołki o takich samych kodach. Z i-tej warstwy wychodzi $3n_i$ krawędzi prowadzących do co najwyżej $3n_i$ wierzchołków. Wierzchołki te możemy wyznaczyć i ponumerować w czasie rzędu n_i. W kodach wierzchołków z i-tej warstwy wierzchołki, do których prowadzą krawędzie możemy reprezentować za pomocą numerów w tej numeracji. Wówczas możemy posortować kody leksykograficznie w czasie rzędu n_i.

Oto fragment programu realizujący sortowanie kodów wierzchołków i-tej warstwy. Zakładamy, że początkowo wszystkie wierzchołki mają zainicjowane pola `numer` na 0, oraz że "kubły" są zainicjowane na puste kolejki.

```
var num, i, j : Word;
  P, Q, W: PElem;
  Kubly : ↑TKolejki; { Kubełki do sortowania leksykograficznego. }
  procedure Przerzuc(var K: TKolejka; var L: PElem);
  { Przerzuca pierwszy element listy L na koniec kolejki K. }
    ⋮
  function ScalKubly(maxnum: Word): PElem;
  { Łączy kolejki z kubełków o numerach [1..maxnum] w jedną listę. }
    ⋮
begin
  num := 0;
  W := Warstwy[i];
  P := W;
  while P ≠ nil do begin
    { Ponumeruj wszystkie wierzchołki, do których prowadzą }
    { krawędzie z i-tej warstwy. }
    for j := 1 to 3 do with V[V[P↑.l].Dol[j]] do
      if numer = 0 then begin
        Inc(num);
        numer := num
      end
    P := P↑.nast
  end
  { Sortuj leksykograficznie po numerach wierzchołków, }
  { do których prowadzą krawędzie z i-tej warstwy. }
  for j := 3 downto 1 do begin
    while W ≠ nil do
      Przerzuc(Kubly↑[V[V[W↑.l].Dol[j]].numer], W);
    W := ScalKubly(num)
  end
  { Sortuj po kolorach wierzchołków. }
  while W ≠ nil do
    Przerzuc(Kubly↑[V[W↑.l].kolor], W);
  Warstwy[i] := ScalKubly(3);
```

```
end
```

Po posortowaniu, takie same kody będą sąsiadować ze sobą — wystarczy więc przejrzeć je w kolejności posortowania i skleić wierzchołki o takich samych kodach.

Sklejając kilka wierzchołków, do wybranego wierzchołka doklejamy pozostałe. Doklejając jeden wierzchołek do drugiego musimy poprawić krawędzie prowadzące do pierwszego z nich — korzystamy tu z list wierzchołków, z których krawędzie prowadzą do danego wierzchołka. Możemy natomiast zaniedbać poprawianie list, na których występują sklejane wierzchołki, gdyż sklejamy wierzchołki w warstwach o coraz większych głębokościach i listy te nie będą już nam potrzebne. Oznaczmy przez w_i liczbę krawędzi prowadzących do i-tej warstwy. Koszt czasowy sklejania wierzchołków z warstwy o głębokości i wynosi $\Theta(n_i + w_i)$. Poniższy fragment kodu skleja odpowiednie wierzchołki.

```
function CzyZgodne(a, b: Word): Boolean;
{ Sprawdza, czy a i b mają te same kody. }
var j: Word;
begin
  CzyZgodne := False;
  if V[a].kolor ≠ V[b].kolor { Muszą mieć ten sam kolor. }
    then exit;
  for j := 1 to 3 do
{ Krawędzie muszą prowadzić do tych samych wierzchołków. }
    if V[a].Dol[j] ≠ V[b].Dol[j] then exit;
  CzyZgodne := True
end
procedure LikwidujKomnate(a, b: Word);
{ Dokleja wierzchołek b do a. }
var
  P: PElem; j: Word;
begin
  P := Gora↑[b];
  while P ≠ nil do begin
    with V[P↑.l] do
      for j := 1 to 3 do
        if Dol[j] = b then Dol[j] := a;
    P := P↑.nast
  end
end
var
  ile, a, b: Word;
  P, Q, W: PElem;
begin
  W := Warstwy[i]; P := W; Q := P↑.nast;
  ile := 1; {Liczba różnych wierzchołków warstwy.}
  while Q ≠ nil do begin {Przejdź posortowaną listę.}
    a := P↑.l; b := Q↑.l; {Wierzchołki do ew. sklejenia.}
    if CzyZgodne(a,b) then
      LikwidujKomnate(a,b)
    else begin {Wierzchołek istotnie różny od dotychczasowych.}
      Inc(ile);
      P := Q;
    end
    Q := Q↑.nast
  end
  P := W;
  while P ≠ nil do begin {Likwiduj numerację.}
```

```
      for j := 1 to 3 do
        V[V[P↑.l].Dol[j]].numer := 0;
      P := P↑.nast
    end
    {ile = liczba różnych wierzchołków warstwy.}
  end
```

Pełną treść przedstawionego rozwiązania można znaleźć na załączonej dyskietce.

Zanalizujmy koszt działania opisanego algorytmu. Z każdym wierzchołkiem i każdą krawędzią jest związana stała ilość informacji. Liczba krawędzi jest liniowa ze względu na liczbę wierzchołków. Tak więc złożoność pamięciowa opisanego algorytmu wynosi $\Theta(n)$, gdzie n jest liczbą wierzchołków.

Podział wierzchołków na warstwy wymaga czasu $\Theta(n)$. Posortowanie wierzchołków i-tej warstwy według ich kodów wymaga czasu $\Theta(n_i)$. Sklejenie wierzchołków z i-tej warstwy o tej samej charakterystyce wymaga czasu $\Theta(n_i + w_i)$. Tak więc łączny czas działania opisanego algorytmu wynosi

$$\Theta\left(n + \Sigma_{i=1}^{h-1} n_i + \Sigma_{i=1}^{h-1}(n_i + w_i)\right)$$

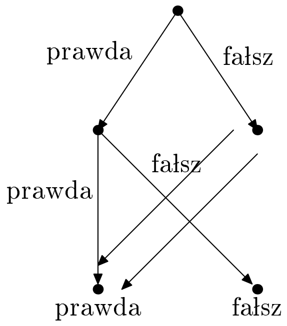
Zauważmy jednak, że $\Sigma_{i=1}^{h-1} n_i = n - 2$ oraz, że $\Sigma_{i=1}^{h-1} w_i < 3n$. Stąd czas działania opisanego algorytmu wynosi $\Theta(n)$.

Binarne diagramy decyzyjne

Opisywane zadanie ma związek z tzw. binarnymi diagramami decyzyjnymi (ang. *binary decision diagrams*, w skrócie BDD) — stosowaną w praktyce strukturą danych. Struktura ta służy do zwięzłego reprezentowania formuł rachunku zdań. Przyjmijmy, że w formułach może się pojawiać k różnych zmiennych zdaniowych: $x_1, \ldots, x_k$. Podobnie jak w przypadku labiryntu Bajtosmoka, BDD ma postać multigrafu skierowanego. Wierzchołki tego multigrafu są podzielone na $k+1$ warstw, o głębokościach 0–k. Warstwę o głębokości 0 tworzą dokładnie dwa wierzchołki, reprezentujące prawdę i fałsz — tak jakby były dwie smocze jamy. Z każdego wierzchołka należącego do pozostałych warstw wychodzą dokładnie po dwie krawędzie, jedna z etykietą "prawda" i jedna z etykietą "fałsz" (zamiast trzech krawędzi ponumerowanych 1, 2 i 3). Krawędzie wychodzące z danego wierzchołka muszą prowadzić do wierzchołków należących do warstwy o głębokości o 1 mniejszej. Warstwę o głębokości k tworzy dokładnie jeden wierzchołek — odpowiednik wejścia do labiryntu. Zauważmy, że wszystkie ścieżki prowadzące z tego wierzchołka do wierzchołków reprezentujących prawdę i fałsz mają długość $k+1$.

W jaki sposób możemy traktować BDD jak formułę rachunku zdań? Dla zadanego wartościowania zmiennych zdaniowych (tzn. przypisania zmiennym zdaniowym wartości prawda/fałsz) formuła rachunku zdań jest albo prawdziwa, albo fałszywa. Aby odczytać, czy dla zadanego wartościowania zmiennych zdaniowych BDD jest prawdziwe, czy fałszywe musimy przejść przez BDD zgodnie z "planem podróży" wyznaczonym przez wartościowanie zmiennych. Przejście zaczynamy od jedynego wierzchołka należącego do warstwy o głębokości k. Między wierzchołkami przechodzimy wzdłuż krawędzi — będąc w wierzchołku należącym do warstwy o głębokości h wybieramy tę krawędź, która ma etykietę równą wartości zmiennej x_h. Wartość BDD dla danego wartościowania jest taka, jak etykieta wierzchołka, do którego docieramy na końcu przejścia.

Poniższy rysunek przedstawia BDD reprezentujące formułę $\neg x_2 \vee (x_1 \wedge x_2)$.



Rys. 1

Zauważmy, że dla każdej formuły rachunku zdań istnieje BDD reprezentujące daną formułę. BDD takie możemy łatwo zbudować. Weźmy pełne drzewo binarne wysokości $k+1$. Krawędzie wychodzące z wierzchołków na lewo etykietujemy prawdą, a wychodzące na prawo fałszem. Zauważmy, że każdemu wartościowaniu zmiennych zdaniowych odpowiada dokładnie jeden liść — ten, do którego prowadzi z korzenia drzewa przejście zgodne z wartościowaniem. Liście drzewa etykietujemy prawdą i fałszem, zgodnie z tym jaka jest wartość formuły dla wartościowania zmiennych odpowiadającego danemu liściowi. Jeżeli skleimy liście mające takie same etykiety (i ew. dodamy jeden wierzchołek, jeżeli wszystkie liście mają takie same etykiety), to uzyskujemy BDD reprezentujące daną formułę.

Zauważmy, że formuła rachunku zdań jest tautologią wtw. gdy w reprezentującym ją BDD żadna krawędź nie prowadzi do wierzchołka reprezentującego fałsz.

Omawiane zadanie odpowiada przekształceniu BDD na równoważne o minimalnej liczbie wierzchołków. Takie BDD stosuje się jako formę zwięzłego reprezentowania formuł rachunku zdań. Warto nadmienić, że dwie formuły rachunku zdań są sobie równoważne wtedy i tylko wtedy, gdy minimalne BDD reprezentujące te formuły są takie same.

TESTY

Do sprawdzania rozwiązań zawodników użyto 14-tu testów. Oto ich krótka charakterystyka:

- LAB1.IN — prosty test badający poprawność rozwiązań, $n = 10$,
- LAB2.IN — test losowy badający poprawność rozwiązań, $n = 50$,
- LAB3.IN — test losowy badający poprawność rozwiązań, $n = 200$,
- LAB4.IN — w tym teście plan labiryntu składa się z 335 warstw, a każda warstwa z 3 wierzchołków: czerwonego, zielonego i niebieskiego, (z wyjątkiem najpłytszej i najgłębszej warstwy, które mają po jednym wierzchołku); z każdego wierzchołka prowadzą krawędzie łączące go z wszystkimi wierzchołkami o głębokości o 1 mniejszej; żadne wierzchołki nie są sklejane ze sobą,
- LAB5.IN — prosty test badający poprawność rozwiązań, $n = 12$,
- LAB6.IN — w tym teście plan labiryntu ma kształt pełnego drzewa wysokości 8, plus smocza jama; wszystkie wierzchołki z jednej warstwy są tego samego koloru, a kolejne warstwy mają kolory dobierane na przemian; w każdej warstwie wszystkie wierzchołki są sklejane ze sobą,
- LAB7.IN — losowy test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, 558 sklejeń wierzchołków,
- LAB8.IN — losowy test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, 51 sklejeń wierzchołków,
- LAB9.IN — losowy test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, żadne wierzchołki nie są sklejane ze sobą,
- LAB10.IN — prosty test badający poprawność rozwiązań, $n = 12$,

- LAB11.IN — test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, dla każdego $i \leqslant n-3$ z wierzchołka nr i prowadzą krawędzie do wierzchołków o numerach $i+1$, $i+2$ i $i+3$, z wierzchołka nr $n-2$ prowadzi jedna krawędź do wierzchołka nr $n-1$ i dwie krawędzie do wierzchołka nr n, a z wierzchołka nr $n-1$ wszystkie krawędzie prowadzą do wierzchołka nr n; żadne wierzchołki nie są sklejane ze sobą,
- LAB12.IN — prosty test badający poprawność rozwiązań, $n = 12$,
- LAB13.IN — test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, każda warstwa składa się z dokładnie jednego wierzchołka, krawędzie wychodzące z każdego wierzchołka prowadzą do (jedynego) wierzchołka o głębokości o jeden mniejszej; żadne wierzchołki nie są sklejane ze sobą,
- LAB14.IN — prosty test badający poprawność rozwiązań, $n = 12$.

W testach nr 4, 9, 11 i 13 nie było żadnych sklejeń wierzchołków. Testy te zostały zgrupowane z prostymi testami badającymi poprawność rozwiązań (4 z 5, 9 z 10, 11 z 12 oraz 13 z 14) — oceniane rozwiązanie otrzymywało punkty za testy z danej grupy tylko wtedy, gdy uzyskiwało punkty za obydwa testy tworzące grupę. Takie grupowanie testów pozwalało wychwycić rozwiązania stwierdzające zawsze, że żadne wierzchołki nie ulegają sklejeniu.

Poniższa tabelka przedstawia wielkości poszczególnych testów oraz wyniki.

Nr testu	**n**	**Wynik**
1	10	6
2	50	33
3	200	106
4	1 001	1 001
5	12	8
6	3 281	9
7	6 000	5 442
8	6 000	5 949
9	6 000	6 000
10	12	7
11	6 000	6 000
12	12	8
13	6 000	6 000
14	12	8