MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

III OLIMPIADA INFORMATYCZNA

1995/1996

WARSZAWA, WROCŁAW – 1996

Autorzy:

Rozdz. 1: prof. dr hab. Maciej M. Sysło
Rozdz. 2–5: Komitet Główny Olimpiady
Rozdz. 6: prof. dr hab. inż. Stanisław Waligórski
Rozdz. 7: Autorzy zadań: dr Piotr Chrząstowski-Wachtel, dr Krzysztof Diks, prof. dr hab. Wojciech Guzicki, mgr Marcin Kubica, dr Wojciech Plandowski, prof. dr hab. Wojciech Rytter
Rozdz. 8: dr Piotr Chrząstowski-Wachtel, dr Krzysztof Diks, prof. dr hab. inż. Stanisław Waligórski – przekłady tekstów zadań

Autorzy programów na dyskietce:

dr Piotr Chrząstowski-Wachtel, dr Krzysztof Diks, mgr Marcin Engel, prof. dr hab. Wojciech Guzicki, Grzegorz Jakacki, Albert Krzymowski, Marek Pawlicki, dr Wojciech Plandowski, prof. dr hab. Wojciech Rytter

Opracowanie i redakcja:

Prof. dr hab. Maciej M. Sysło

Skład komputerowy:

mgr Magdalena Kraszewska

ISBN 83–902611–7–0

Nakład 2000 egz.

Druk i oprawa: Zakład Poligraficzny Kuratorium Oświaty
ul. Paryska 25, 03-945 Warszawa, tel./fax 617 60 87

SPIS TREŚCI

1. WSTĘP

Zawody informatyczne dla młodzieży mają u nas w kraju już dość długą historię. Początkowo były to przede wszystkim zawody o lokalnym zasięgu, organizowane przez nauczycieli-entuzjastów i niektóre organizacje wspierające szkoły. Takie zawody dla młodzieży szkolnej są nadal organizowane, głównie we współpracy z Wojewódzkimi Ośrodkami Metodycznymi.

W grudniu 1993 roku, po spełnieniu wymogów zarządzenia ministra edukacji narodowej w sprawie organizacji olimpiad przedmiotowych, Instytut Informatyki Uniwersytetu Wrocławskiego powołał Olimpiadę Informatyczną.

W roku szkolnym 1995/1996 odbyła się już III Olimpiada Informatyczna. Oddajemy do rąk Czytelników szczegółową relację z jej przebiegu oraz z olimpiady międzynarodowej, w której brali udział zdobywcy czterech pierwszych miejsc w olimpiadzie krajowej.

Ponieważ informacje o Olimpiadzie Informatycznej nie dotarły jeszcze do wszystkich szkół, zawarliśmy w tym opracowaniu również oficjalne dokumenty Komitetu Głównego: Regulamin Olimpiady Informatycznej i Zasady organizacji zawodów w 1995/1996 roku. Te dwa dokumenty zawierają wiele informacji ważnych dla uczestników olimpiad a dotyczących organizacji zawodów, formalnych wymogów stawianych rozwiązaniom zadań oraz wyróżnień i nagród przyznawanych uczniom i nauczycielom.

Uczniów interesujących się informatyką, w tym potencjalnych uczestników Olimpiady Informatycznej, oraz ich nauczycieli najbardziej zainteresują zapewne szczegółowe opisy metod rozwiązywania zadań oraz omówienia rozwiązań podanych przez uczniów, napisane przez autorów zadań.

Podstawowym elementem rozwiązania każdego zadania Olimpiady jest program, realizujący odpowiednio dobrany algorytm rozwiązywania postawionego zadania. Inne elementy rozwiązań, jak opis algorytmu i dokumentacja programu, pełnią rolę pomocniczą – pozytywną ocenę otrzymuje się wtedy, gdy program jest, poprawnie rozwiąznje zadanie i robi to możliwie efektywanie dzięki użyciu właściwego algorytmu.

O tym, jak dobierać algorytmy, piszą autorzy zadań, starając się zwrócić uwagę na podstawowe czynniki decydujące o jakości rozwiązań. Powinno to

skłonić zainteresowanych uczniów do dalszego pogłębiania swojej wiedzy i umiejętności oraz sięgnięcia po podręczniki poświęcone algorytmice, algorytmom i strukturom danych i metodom obliczeniowym z różnych dziedzin. Listę polecanych książek zamieszczamy na końcu tych materiałów. W rozwiązaniach zadań odwołujemy się do tych książek podając numer na tej liście zamknięty w nawiasy kwadratowe [ ]*.

Nowością tych materiałów jest załączona dyskietka, na której w kartotekach poszczególnych zadań zostały umieszczone programy w języku Pascal rozwiązujące zadania oraz wszystkie testy stosowane przez Jury do sprawdzania rozwiązań.

W materiałach jest zamieszczona również relacja z międzynarodowej Olimpiady Informatycznej (rozdz. 6), w której brała udział ekipa Polski. Rozdział 8 zawiera teksty zadań z tych zawodów. Start naszej ekipy w tegorocznych zawodach międzynarodowych zakończył się dużym sukcesem. Nasi uczniowie zdobyli dwa złote medale, jeden srebrny i jeden brązowy, a w klasyfikacji państw Polska zajęła czwarte miesjce.

Staraliśmy się, by to opracowanie trafiło do rąk Czytelników jeszcze przed rozesłaniem pierwszych zadań IV Olimpiady Informatycznej i stanowiło w miarę pełne źródło informacji o sposobie przeprowadzania zawodów Olimpiady oraz o charakterze zadań i ich rozwiązaniach. W pośpiechu nie uniknęliśmy zapewne potknięć i usterek – za ich wskazanie będziemy wdzięczni tak, by relacje z następnych Olimpiad były od nich wolne.

* Materiały z kolejnych olimpiad informatycznych można nabyć w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów (ul. Raszyńska 8/10, 02-026 Warszawa, tel. 22-224019) i w Instytucie Informatyki Uniwersytetu Wrocławskiego (ul. Przesmyckiego 20, 51-151 Wrocław, tel. 71-251271). Pojedyncze egzemplarze posiadają również wojewódzcy koordynatorzy edukacji informatycznej oraz nauczyciele informatyki w szkołach.

2. REGULAMIN OLIMPIADY INFORMATYCZNEJ*

§ 1. Wstęp

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, który jest organizatorem Olimpiady zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku. W organizacji Olimpiady Instytut będzie współdziałał ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§ 2. Cele Olimpiady Informatycznej

1. Stworzenie motywacji dla zainteresowania nauczycieli i uczniów nowymi metodami informatyki.
2. Rozszerzanie współdziałania nauczycieli akademickich z nauczycielami szkół w kształceniu młodzieży uzdolnionej.
3. Stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej.
4. Kształtowanie umiejętności samodzielnego zdobywania i rozszerzania wiedzy informatycznej.
5. Stwarzanie młodzieży możliwości szlachetnego współzawodnictwa w rozwijaniu swoich uzdolnień, a nauczycielom – warunków twórczej pracy z młodzieżą.
6. Wyłanianie reprezentacji Rzeczypospolitej Polskiej na Międzynarodową Olimpiadę Informatyczną.

---

* W dniu 27.06 1994 roku Komitet Główny Olimpiady wprowadził zmiany w regulaminie, związane z utworzeniem Komitetów Okręgowych Olimpiady, których głównym zadaniem jest organizacja zawodów II stopnia (zob. §3, pkt 11; §4, pkt 5; §5 oraz rozdz. 5). Publikujemy obowiązującą wersję regulaminu.

§ 3. Organizacja Olimpiady

1. Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.
2. Olimpiada Informatyczna jest trójstopniowa.
3. W Olimpiadzie Informatycznej mogą brać indywidualnie udział uczniowie wszystkich typów szkół średnich dla młodzieży (z wyjątkiem szkół policealnych).
4. W Olimpiadzie mogą również uczestniczyć – za zgodą Komitetu Głównego – uczniowie szkół podstawowych.
5. Liczbę i treść zadań na każdy stopień zawodów ustala Komitet Główny, wybierając je drogą głosowania spośród zgłoszonych projektów.
6. Integralną częścią rozwiązania zadań zawodów I, II i III stopnia jest program napisany na komputerze zgodnym ze standardem IBM PC, w języku programowania wysokiego poziomu (takim na przykład, jak: Pascal, Basic, Logo, Lisp, C, C++, Prolog). Rozwiązania w asemblerze nie będą brane pod uwagę.
7. Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz nadesłaniu rozwiązań pod adresem Komitetu Olimpiady Informatycznej w podanym terminie.
8. Liczbę uczestników zakwalifikowanych do zawodów II i III stopnia ustala Komitet Główny i podaje ją w „Zasadach organizacji zawodów" na dany rok szkolny.
9. O zakwalifikowaniu uczestnika do zawodów kolejnego stopnia decyduje Komitet Główny na podstawie rozwiązań zadań niższego stopnia. Oceny zadań dokonuje jury powołane przez Komitet i pracujące pod kierownictwem przewodniczącego Komitetu i sekretarza naukowego Olimpiady. Zasady oceny ustala Komitet na podstawie propozycji zgłaszanych przez kierownictwo jury oraz autorów i recenzentów zadań. Wyniki proponowane przez jury podlegają zatwierdzeniu przez Komitet.
10. Komitet Główny Olimpiady kwalifikuje do zawodów II i III stopnia odpowiednią liczbę uczestników, których rozwiązania zadań stopnia niższego ocenione zostaną najwyżej.
11. Zawody II stopnia są przeprowadzane przez Komitety Okręgowe Olimpiady. Pierwsze sprawdzenie rozwiązań jest dokonywane bezpośrednio po zawodach przez znajdującą się na miejscu część jury. Ostateczną ocenę prac ustala jury w pełnym składzie po powtórnym sprawdzeniu prac.
12. Zawody II i III stopnia polegają na samodzielnym rozwiązaniu przez uczestników Olimpiady zakwalifikowanych do tych zawodów zadań przygotowa-

nych dla danego stopnia w ciągu dwóch sesji przeprowadzanych w różnych dniach w warunkach kontrolowanej samodzielności.

13. Prace zespołowe, niesamodzielne lub nieczytelne nie będą brane pod uwagę.

§ 4. Komitet Główny Olimpiady Informatycznej

1. Komitet Główny Olimpiady Informatycznej, zwany dalej Komitetem, powoływany przez organizatora na kadencję trzyletnią, jest odpowiedzialny za poziom merytoryczny i organizację zawodów. Komitet składa corocznie organizatorowi sprawozdanie z przeprowadzonych zawodów.
2. Członkami Komitetu mogą być pracownicy naukowi, nauczyciele i pracownicy oświaty związani z kształceniem informatycznym.
3. Komitet wybiera ze swego grona prezydium, które podejmuje decyzje w nagłych sprawach pomiędzy posiedzeniami Komitetu. W skład prezydium wchodzą przewodniczący, wiceprzewodniczący, sekretarz naukowy i kierownik organizacyjny.
4. Komitet Główny może w czasie swojej kadencji dokooptować nowych członków.
5. Komitet Główny powołuje Komitety Okręgowe Olimpiady w miarę powiększania się liczby uczestników zawodów i powstawania odpowiednich warunków organizacyjnych.
6. Komitet Główny powołuje corocznie Komitet Honorowy spośród sponsorów przyczyniających się do rozwoju Olimpiady, który podejmuje decyzje dotyczące uhonorowania laureatów i finalistów nagrodami rzeczowymi.
7. Komitet Główny Olimpiady Informatycznej:
   a) opracowuje szczegółowe „Zasady organizacji zawodów", które są ogłaszane razem z treścią zadań zawodów I stopnia Olimpiady,
   b) ustala treść tematów zadań na wszystkie stopnie Olimpiady,
   c) powołuje jury Olimpiady, które jest odpowiedzialne za sprawdzenie zadań,
   d) udziela wyjaśnień w sprawach dotyczących Olimpiady,
   e) ustala listy uczestników zawodów II i III stopnia,
   f) ustala listy laureatów i uczestników wyróżnionych oraz kolejność lokat,
   g) przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady,
   h) ustala kryteria wyłaniania uczestników uprawnionych do startu w Międzynarodowej Olimpiadzie Informatycznej i publikuje je w „Zasadach organizacji zawodów" oraz ustala ostateczną listę reprezentacji,
   i) zatwierdza liczbę etatów biura Olimpiady na wniosek kierownika organizacyjnego, który odpowiada za sprawne działanie biura.

8. Decyzje Komitetu zapadają zwykłą większością głosów uprawnionych przy obecności przynajmniej połowy członków Komitetu Głównego. W przypadku równej liczby głosów decyduje głos przewodniczącego obrad.
9. Posiedzenia Komitetu, na których ustala się treść zadań Olimpiady są tajne. Przewodniczący obrad może zarządzić tajność obrad także w innych uzasadnionych przypadkach.
10. Decyzje Komitetu we wszystkich sprawach dotyczących zadań i uczestników są ostateczne.
11. Komitet dysponuje funduszem Olimpiady za pośrednictwem kierownika organizacyjnego Olimpiady.
12. Komitet ma siedzibę w Warszawie w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów Kuratorium Oświaty w Warszawie. Ośrodek wspiera Komitet we wszystkich działaniach organizacyjnych zgodnie z Deklaracją przekazaną organizatorowi.
13. Pracami Komitetu kieruje przewodniczący, a w jego zastępstwie lub z jego upoważnienia wiceprzewodniczący.
14. Przewodniczący:
    a) czuwa nad całokształtem prac Komitetu,
    b) zwołuje posiedzenia Komitetu,
    c) przewodniczy tym posiedzeniom,
    d) reprezentuje Komitet na zewnątrz,
    e) czuwa nad prawidłowością wydatków związanych z organizacją i przeprowadzeniem Olimpiady oraz zgodnością działalności Komitetu z przepisami.
15. Komitet prowadzi archiwum akt Olimpiady przechowując w nim:
    a) zadania Olimpiady,
    b) rozwiązania zadań Olimpiady przez okres 2 lat,
    c) rejestr wydanych zaświadczeń i dyplomów laureatów,
    d) listy laureatów i ich nauczycieli,
    e) dokumentację statystyczną i finansową.
14. W jawnych posiedzeniach Komitetu mogą brać udział przedstawiciele organizacji wspierających jako obserwatorzy z głosem doradczym.

§ 5. Komitety Okręgowe

1. Komitet Okręgowy składa się z przewodniczącego, jego zastępcy, sekretarza i co najmniej dwóch członków mianowanych przez Komitet Główny na okres swojej kadencji.
2. Zmiany w składzie Komitetu Okręgowego są każdorazowo dokonywane przez Komitet Główny.

3. Zadaniem Komitetów okręgowych jest organizacja zawodów II stopnia oraz popularyzacja Olimpiady.
4. Przewodniczący (albo jego zastępca) oraz sekretarz Komitetu Okręgowego mogą uczestniczyć w obradach Komitetu Głównego z prawem głosu.

§ 6. Przebieg Olimpiady

1. Komitet Główny rozsyła do młodzieżowych szkół średnich oraz Kuratoriów Oświaty i Koordynatorów Edukacji Informatycznej treść zadań I stopnia wraz z „Zasadami organizacji zawodów".
2. W czasie rozwiązywania zadań w zawodach II i III stopnia każdy uczestnik ma do swojej dyspozycji komputer zgodny ze standardem IBM PC.
3. Rozwiązywanie zadań Olimpiady w zawodach II i III stopnia jest poprzedzone jednodniowymi sesjami próbnymi umożliwiającymi zapoznanie się uczestników z warunkami organizacyjnymi i technicznymi Olimpiady.
4. Komitet Główny Olimpiady Informatycznej zawiadamia uczestnika oraz dyrektora jego szkoły o zakwalifikowaniu do zawodów stopnia II i III, podając jednocześnie miejsce i termin zawodów.
5. Uczniowie powołani do udziału w zawodach II i III stopnia są zwolnieni z zajęć szkolnych na czas niezbędny do udziału w zawodach, a także otrzymują bezpłatne zakwaterowanie i wyżywienie oraz zwrot kosztów przejazdu.

§ 7. Uprawnienia i nagrody

1. Uczestnicy zawodów stopnia II i III otrzymują nagrody rzeczowe.
2. Uczestnicy zawodów II stopnia, których wyniki zostały uznane przez Komitet Główny Olimpiady za wyróżniające, otrzymują najwyższą ocenę z informatyki na zakończenie nauki w klasie, do której uczęszczają.
3. Uczestnicy Olimpiady, którzy zostali zakwalifikowani do zawodów III stopnia są zwolnieni z egzaminu z przygotowania zawodowego z przedmiotu informatyka oraz (zgodnie z zarządzeniem nr 35 Ministra Edukacji Narodowej z dnia 30 listopada 1991 r.) z części ustnej egzaminu dojrzałości z przedmiotu informatyka, jeżeli w klasie do której uczęszczał zawodnik był realizowany rozszerzony, indywidualnie zatwierdzony przez MEN program nauczania tego przedmiotu.
4. Laureaci zawodów III stopnia, a także finaliści są zwolnieni w części lub w całości z egzaminów wstępnych do szkół wyższych – na mocy uchwał senatów poszczególnych uczelni, podjętych zgodnie z przepisami ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65 poz. 385) – o ile te uchwały nie stanowią inaczej.

5. Zaświadczenia o uzyskanych uprawnieniach wydają uczestnikom Komitet Główny i komitety okręgowe. Zaświadczenia podpisuje przewodniczący Komitetu. Komitet prowadzi rejestr wydanych zaświadczeń.
6. Nauczyciel (opiekun naukowy), którego praca przy przygotowaniu uczestnika Olimpiady zostanie oceniona przez Komitet Główny jako wyróżniająca otrzymuje nagrodę wypłacaną z budżetu Olimpiady.
7. Komitet Główny Olimpiady przyznaje wyróżniającym się aktywnością członkom Komitetu nagrody pieniężne z funduszu Olimpiady.

§ 8. Finansowanie Olimpiady

1. Komitet Główny będzie się ubiegał o dotację z budżetu państwa, składając wniosek w tej sprawie do Ministra Edukacji Narodowej i przedstawiając przewidywany plan finansowy organizacji Olimpiady na dany rok. Komitet będzie także zabiegał o pozyskanie dotacji z innych organizacji wspierających.

§ 9. Przepisy końcowe

1. Koordynatorzy Edukacji Informatycznej i dyrektorzy szkół mają obowiązek dopilnowania, aby wszystkie wytyczne oraz informacje dotyczące Olimpiady zostały podane do wiadomości uczniów.
2. Wyniki zawodów I stopnia Olimpiady są tajne do czasu ustalenia listy uczestników zawodów II stopnia. Wyniki zawodów II stopnia są tajne do czasu ustalenia listy uczestników zawodów III stopnia Olimpiady.
3. Komitet Główny zatwierdza sprawozdanie z przeprowadzonej Olimpiady w ciągu 2 miesięcy po jej zakończeniu i przedstawia je organizatorowi i Ministerstwu Edukacji Narodowej.
4. Niniejszy regulamin może być zmieniony przez Komitet Główny tylko przed rozpoczęciem kolejnej edycji zawodów Olimpiady po zatwierdzeniu zmian przez organizatora i uzyskaniu aprobaty Ministerstwa Edukacji Narodowej.

3. ZASADY ORGANIZACJI ZAWODÓW W ROKU SZKOLNYM 1995/96

Podstawowym aktem prawnym dotyczącym Olimpiady jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach oświaty*. „Zasady organizacji zawodów" są uzupełnieniem tego regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 1995/96.

§ 1. Wstęp

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, który jest organizatorem Olimpiady zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku.

§ 2. Organizacja Olimpiady

1. Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.
2. Olimpiada Informatyczna jest trójstopniowa.
3. Olimpiada Informatyczna jest przeznaczona dla uczniów wszystkich typów szkół średnich dla młodzieży (z wyjątkiem szkół policealnych i wyższych uczelni). W Olimpiadzie mogą również uczestniczyć – za zgodą Komitetu Głównego – uczniowie szkół podstawowych.
4. Integralną częścią rozwiązania każdego z zadań zawodów I, II i III stopnia jest program napisany na komputerze zgodnym ze standardem IBM PC, w języku programowania wysokiego poziomu (takim na przykład, jak: Pascal, Basic, Logo, C, C++). Rozwiązania w asemblerze nie będą brane pod uwagę.

* Pełny tekst Regulaminu Olimpiady Informatycznej jest zamieszczony w rozdziale 2.

5. Zawody I stopnia mają charakter otwarty i polegają na samodzielnym i indywidualnym rozwiązywaniu zadań i nadesłaniu rozwiązań w podanym terminie.
6. Zawody II i III stopnia polegają na samodzielnym rozwiązywaniu zadań w ciągu dwóch sesji przeprowadzanych w różnych dniach w warunkach kontrolowanej samodzielności.
7. Do zawodów II stopnia zostanie zakwalifikowanych 100 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia – 40 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 10%.
8. Podjęte przez Komitet Główny decyzje o zakwalifikowaniu uczestników do zawodów kolejnego stopnia, przyznanych miejscach i nagrodach oraz składzie polskiej reprezentacji na Międzynarodową Olimpiadę Informatyczną są ostateczne.

§ 3. Wymagania dotyczące rozwiązań zadań zawodów I stopnia

1. Zawody I stopnia polegają na samodzielnym i indywidualnym rozwiązywaniu zadań eliminacyjnych (niekoniecznie wszystkich) i nadesłaniu rozwiązań pocztą, **przesyłką poleconą**, pod adresem:

Olimpiada Informatyczna
Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
ul. Raszyńska 8/10, 02-026 Warszawa
tel. (0-22) 22-40-19

   w nieprzekraczalnym terminie nadania do poniedziałku 20.11.1995 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki. Rozwiązania dostarczane w inny sposób nie będą przyjmowane.
2. Prace niesamodzielne lub zbiorowe nie będą brane pod uwagę.
3. Rozwiązanie każdego zadania składa się z:
   a) programu (tylko jednego) na dyskietce w postaci źródłowej i skompilowanej (w Logo – tylko w postaci źródłowej),
   b) wydrukowanego tekstu tego programu w postaci źródłowej,
   c) opisu algorytmu rozwiązania zadania z uzasadnieniem jego poprawności.
4. Uczestnik przysyła jedną dyskietkę, oznaczoną jego imieniem i nazwiskiem, nadającą się do uruchomienia na komputerze IBM PC i zawierającą:
   — spis zawartości dyskietki w pliku nazwanym SPIS.TRC,
   — wszystkie programy w postaci źródłowej i skompilowanej (w Logo – tylko w postaci źródłowej).

Imię i nazwisko uczestnika powinno być podane w komentarzu na początku każdego programu.

5. Wszystkie nadsyłane teksty powinny być drukowane (lub czytelnie pisane) jednostronnie na kartkach formatu A4. Każda kartka powinna mieć kolejny numer i być opatrzona pełnym imieniem i nazwiskiem autora. Na pierwszej stronie nadsyłanej pracy każdy uczestnik Olimpiady podaje następujące dane:
   — imię i nazwisko,
   — datę i miejsce urodzenia,
   — dokładny adres zamieszkania i ewentualnie numer telefonu,
   — nazwę, adres i numer telefonu szkoły oraz klasę, do której uczęszcza,
   — nazwę i numer wersji użytego języka programowania,
   — opis konfiguracji komputera, na którym rozwiązał zadania.
6. Nazwy plików z programami w postaci źródłowej powinny mieć jako rozszerzenie co najwyżej trzyliterowy skrót nazwy tego języka programowania, na przykład:

Pascal	PAS
Basic	BAS
Logo	LOG
C	C
C++	CPP

7. Opcje kompilatora powinny być częścią tekstu programu. Zaleca się stosowanie opcji standardowych.

§ 4. Uprawnienia i nagrody

1. Uczestnicy zawodów II stopnia, których wyniki zostały uznane przez Komitet Główny Olimpiady za wyróżniające, otrzymują najwyższą ocenę z informatyki na zakończenie nauki w klasie, do której uczęszczają.
2. Uczestnicy Olimpiady, którzy zostali zakwalifikowani do zawodów III stopnia, są zwolnieni z egzaminu dojrzałości (zgodnie z zarządzeniem nr 29 Ministra Edukacji Narodowej z dnia 30 listopada 1991 r.) lub z egzaminu z przygotowania zawodowego z przedmiotu informatyka. Zwolnienie jest równoznaczne z wystawieniem oceny najwyższej.
3. Laureaci i finaliści Olimpiady są zwolnieni w części lub w całości z egzaminów wstępnych do szkół wyższych na mocy uchwał senatów poszczególnych uczelni, podjętych zgodnie z przepisami ustawy z dnia 12 września 1990 roku o szkolnictwie wyższym (Dz.U. nr 65, poz. 385), o ile te uchwały nie stanowią inaczej.
4. Zaświadczenia o uzyskanych uprawnieniach wydaje uczestnikom Komitet Główny.

5. Komitet Główny ustala skład reprezentacji Polski na VIII Międzynarodową Olimpiadę Informatyczną w 1996 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na VIII Międzynarodową Olimpiadę Informatyczną.
6. Nauczyciel (opiekun naukowy), który przygotował laureata Olimpiady Informatycznej, otrzymuje nagrodę przyznawaną przez Komitet Główny Olimpiady.
7. Uczestnicy zawodów II i III stopnia otrzymują nagrody rzeczowe.

§ 5. Przepisy końcowe

1. Koordynatorzy edukacji informatycznej i dyrektorzy szkół mają obowiązek dopilnowania, aby wszystkie informacje dotyczące Olimpiady zostały podane do wiadomości uczniów.
2. Komitet Główny Olimpiady Informatycznej zawiadamia wszystkich uczestników zawodów I i II stopnia o ich wynikach. Każdy uczestnik, który przeszedł do zawodów wyższego stopnia oraz dyrektor jego szkoły otrzymują informację o miejscu i terminie następnych zawodów.
3. Uczniowie zakwalifikowani do udziału w zawodach II i III stopnia są zwolnieni z zajęć szkolnych na czas niezbędny do udziału w zawodach, a także otrzymują bezpłatne zakwaterowanie i wyżywienie oraz zwrot kosztów przejazdu.

Uwaga. W materiałach rozsyłanych do szkół, po „Zasadach organizacji zawodów" zostały umieszczone treści czterech zadań zawodów I stopnia, a po nich – następujące „Wskazówki dla uczestników":

1. Przeczytaj uważnie nie tylko teksty zadań, ale i treść „Zasad organizacji zawodów".
2. Przestrzegaj dokładnie warunków określonych w tekście zadania, w szczególności wszystkich reguł dotyczących nazw plików.
3. Twój program powinien czytać dane z pliku i zapisywać wyniki do pliku o podanych w treści zadania nazwach. Nie można podejmować żadnych prób konwersacji z użytkownikiem.
4. Dane testowe są zawsze zapisywane bezbłędnie, zgodnie z warunkami zadania i podaną specyfikacją wejścia. Twój program nie musi tego sprawdzać. Nie przyjmuj żadnych założeń, które nie wynikają z treści zadania.
5. Staraj się dobrać taką metodę rozwiązania zadania, która jest nie tylko poprawna, ale daje wyniki w jak najkrótszym czasie.

6. Ocena za rozwiązanie zadania jest określana na podstawie wyników testowania programu i uwzględnia poprawność oraz efektywność metody rozwiązania użytej w programie.

4. REGULAMIN PRZEPROWADZENIA ZAWODÓW II i III STOPNIA

1. Zawody II i III stopnia Olimpiady Informatycznej polegają na samodzielnym rozwiązaniu zadań w ciągu pięciogodzinnych sesji, w dwóch różnych dniach.
2. Rozwiązywanie zadań konkursowych w zawodach II i III stopnia jest poprzedzone jednodniową sesją próbną umożliwiającą uczestnikom zapoznanie się z warunkami organizacyjnymi i technicznymi Olimpiady. Wyniki sesji próbnej nie liczą się do klasyfikacji.
3. W czasie rozwiązywania zadań konkursowych każdy uczestnik ma do swojej dyspozycji komputer zgodny ze standardem IBM PC. Zawodnikom wolno korzystać wyłącznie ze sprzętu dostarczonego przez organizatora. Stanowiska są przydzielane losowo.
4. Na sprawdzenie kompletności oprogramowania i poprawności konfiguracji sprzętu jest przeznaczone pół godziny przed rozpoczęciem sesji próbnej. W tym czasie wszystkie zauważone braki powinny zostać usunięte. Jeżeli nie wszystko uda się poprawić w tym czasie, rozpoczęcie sesji próbnej w tej sali może się opóźnić.
5. W przypadku stwierdzenia przez jury awarii sprzętu w czasie zawodów, termin zakończenia pracy przez uczestnika zostaje przedłużony o tyle, ile trwało usunięcie awarii.
6. Jedyną dopuszczalną literaturą w czasie trwania zawodów są firmowe opisy oprogramowania. Nie można korzystać z żadnych innych książek ani pomocy, takich jak: dyski, notatki, wydruki.
7. W czasie pięciogodzinnej sesji:
   a) W ciągu pierwszych 30 minut uczestnik może zadawać w formie pisemnej pytania kierowane do jurorów, na które otrzymuje na piśmie jedną z trzech odpowiedzi: „tak", „nie", „bez odpowiedzi".
   b) Jakikolwiek inny sposób komunikowania się z członkami jury co do treści i sposobów rozwiązywania zadań nie jest dopuszczalny.

c) Komunikowanie się z innymi uczestnikami Olimpiady jest podczas rozwiązywania zadań zabronione, pod rygorem dyskwalifikacji.

d) Każdy uczestnik dostaje do dyspozycji jedną oznakowaną dyskietkę i korzystanie z innych dyskietek, poza przypadkiem omówionym w punkcie e), jest zabronione pod rygorem dyskwalifikacji.

e) Uczestnicy, którzy chcą skorzystać z drukarki otrzymują od Komisji Technicznej dyskietkę, na którą nagrywają plik do wydrukowania.*

f) Po upływie wyznaczonego czasu wszelkie czynności uczestnika przy jego komputerze są wzbronione i przekroczenie tego zakazu powoduje dyskwalifikację uczestnika.

g) Gdy sesja zbliża się ku końcowi zawodnik powinien skopiować ostateczne rozwiązania zadań na dyskietkę. Nie będzie żadnego dodatkowego czasu na kopiowanie po zakończeniu sesji. Sprawdzeniu podlegają tylko programy zapisane na dyskietce.

f) Na dyskietce może być tylko jedno rozwiązanie każdego zadania, w skład którego wchodzą odpowiednie pliki źródłowe i wykonywalne. Podstawą oceny będzie plik ???.EXE. W przypadku odstępstwa od tego punktu regulaminu ocenie podlega plik zapisany najpóźniej.

8. Każdy zawodnik powinien sporządzić krótki opis użytej przez niego metody rozwiązania zadania uzupełniony ewentualnie krótkim uzasadnieniem jej poprawności. Opis jest źródłem informacji dla jury o użytej przez zawodnika metodzie.

9. Każda kartka przekazywanych jury materiałów musi być podpisana imieniem i nazwiskiem uczestnika Olimpiady. Dane te należy też podać w komentarzu na początku każdego programu.

10. Każdego dnia po upłynięciu czasu przeznaczonego na rozwiązywanie zadań, zawodnik jest zobowiązany do odbycia rozmowy z członkiem jury. Celem rozmowy jest sprawdzenie rozwiązania na testach dostarczonych przez jury oraz ewentualne uściślenie użytej w programie metody rozwiązywania.

11. W sprawach spornych ostateczne decyzje podejmuje Jury Odwoławcze, w skład którego wchodzą: Przewodniczący Komitetu Okręgowego, członek Komitetu Okręgowego i przedstawiciel Prezydium Komitetu Głównego – w zawodach II stopnia lub dwaj członkowie Prezydium Komitetu Głównego oraz jeden członek jury nie będący stroną w kwestii spornej podlegającej odwołaniu.

* W zawodach II stopnia odbywających się we Wrocławiu, dzięki podłączeniu wszystkich komputerów do sieci Novell, zawodnicy mogli ponadto drukować bezpośrednio z komputerów, na których pracowali.

5. SPRAWOZDANIE Z PRZEBIEGU III OLIMPIADY INFORMATYCZNEJ

Olimpiada Informatyczna została powołana 10 grudnia 1993 roku przez Instytut Informatyki Uniwersytetu Wrocławskiego zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku (zob. Akt powołania w [1], rozdz. 2).

5.1. Organizacja zawodów

Olimpiada Informatyczna jest trójstopniowa. Integralną częścią rozwiązania każdego zadania zawodów I, II i III stopnia jest program napisany na komputerze zgodnym ze standardem IBM PC, w języku programowania wysokiego poziomu (takim na przykład, jak: Pascal, Basic, Logo, Lisp, C, C++, Prolog). Zawody I stopnia miały charakter otwartego konkursu przeprowadzonego dla uczniów wszystkich typów szkół młodzieżowych.

Dnia 10 października 1995 roku rozesłano plakaty zawierające zasady organizacji zawodów I stopnia (zob. rozdz. 3) oraz zestaw 4 zadań konknrsowych do 3258 szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 23 października 1995 roku. Ostatecznym terminem nadsyłania prac konkursowych był 20 listopada 1995 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w trzech okręgach w dniach 1–3.02.1996 roku, natomiast zawody III stopnia odbyły się w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie w dniach 25–29.03.1996 roku.

Uroczystość zakończenia III Olimpiady Informatycznej odbyła się w dniu 29 marca 1996 roku w Auli Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie.

5.2. Skład osobowy Komitetów Olimpiady Informatycznej

5.2.1. Komitet Główny

Prezydium:

przewodniczący

prof. dr hab. inż. Stanisław Waligórski, Uniwersytet Warszawski,

zastępca przewodniczącego

prof. dr hab. Maciej M. Sysło, Uniwersytet Wrocławski,

sekretarz naukowy

dr Andrzej Walat, OEIiZK,

kierownik organizacyjny

Tadeusz Kuran, OEIiZK,

sekretarz

mgr Krystyna Kominek, II L.O. im. St. Batorego w Warszawie.

Członkowie:

prof. dr hab. Jacek Błażewicz, Politechnika Poznańska,
prof. dr hab. Jan Madey, Uniwersytet Warszawski,
prof. dr hab. Andrzej W. Mostowski, Uniwersytet Gdański,
prof. dr hab. Wojciech Rytter, Uniwersytet Warszawski,
dr Piotr Chrząstowski-Wachtel, Uniwersytet Warszawski,
dr Krzysztof Diks, Uniwersytet Warszawski,
dr Bolesław Wojdyło, Uniwersytet Mikołaja Kopernika w Toruniu,
mgr Wojciech Complak, Politechnika Poznańska,
mgr Jerzy Dałek, Ministerstwo Edukacji Narodowej,
mgr Marcin Kubica, Uniwersytet Warszawski,
mgr Krzysztof J. Święcicki, Ministerstwo Edukacji Narodowej.

Siedzibą Komitetu Głównego Olimpiady Informatycznej jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie przy ul. Raszyńskiej 8/10 (w skrócie OEIiZK).

Komitet Główny odbył 8 posiedzeń, a Prezydium – 6; protokoły z tych posiedzeń znajdują się w sekretariacie Olimpiady. Dnia 26 stycznia 1996 r. odbyło się jednodniowe seminarium dla członków komitetów okręgowych, poświęcone organizacji zawodów II stopnia w okręgach.

5. SPRAWOZDANIE Z PRZEBIEGU III OLIMPIADY INFORMATYCZNEJ

Olimpiada Informatyczna została powołana 10 grudnia 1993 roku przez Instytut Informatyki Uniwersytetu Wrocławskiego zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku (zob. Akt powołania w [1], rozdz. 2).

5.1. Organizacja zawodów

Olimpiada Informatyczna jest trójstopniowa. Integralną częścią rozwiązania każdego zadania zawodów I, II i III stopnia jest program napisany na komputerze zgodnym ze standardem IBM PC, w języku programowania wysokiego poziomu (takim na przykład, jak: Pascal, Basic, Logo, Lisp, C, C++, Prolog). Zawody I stopnia miały charakter otwartego konkursu przeprowadzonego dla uczniów wszystkich typów szkół młodzieżowych.

Dnia 10 października 1995 roku rozesłano plakaty zawierające zasady organizacji zawodów I stopnia (zob. rozdz. 3) oraz zestaw 4 zadań konkursowych do 3258 szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 23 października 1995 roku. Ostatecznym terminem nadsyłania prac konkursowych był 20 listopada 1995 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w trzech okręgach w dniach 1–3.02.1996 roku, natomiast zawody III stopnia odbyły się w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie w dniach 25–29.03.1996 roku.

Uroczystość zakończenia III Olimpiady Informatycznej odbyła się w dniu 29 marca 1996 roku w Auli Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie.

5.2. Skład osobowy Komitetów Olimpiady Informatycznej

5.2.1. Komitet Główny

Prezydium:

przewodniczący

prof. dr hab. inż. Stanisław Waligórski, Uniwersytet Warszawski,

zastępca przewodniczącego

prof. dr hab. Maciej M. Sysło, Uniwersytet Wrocławski,

sekretarz naukowy

dr Andrzej Walat, OEIiZK,

kierownik organizacyjny

Tadeusz Kuran, OEIiZK,

sekretarz

mgr Krystyna Kominek, II L.O. im. St. Batorego w Warszawie.

Członkowie:

prof. dr hab. Jacek Błażewicz, Politechnika Poznańska,
prof. dr hab. Jan Madey, Uniwersytet Warszawski,
prof. dr hab. Andrzej W. Mostowski, Uniwersytet Gdański,
prof. dr hab. Wojciech Rytter, Uniwersytet Warszawski,
dr Piotr Chrząstowski-Wachtel, Uniwersytet Warszawski,
dr Krzysztof Diks, Uniwersytet Warszawski,
dr Bolesław Wojdyło, Uniwersytet Mikołaja Kopernika w Toruniu,
mgr Wojciech Complak, Politechnika Poznańska,
mgr Jerzy Dałek, Ministerstwo Edukacji Narodowej,
mgr Marcin Kubica, Uniwersytet Warszawski,
mgr Krzysztof J. Święcicki, Ministerstwo Edukacji Narodowej.

Siedzibą Komitetu Głównego Olimpiady Informatycznej jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie przy ul. Raszyńskiej 8/10 (w skrócie OEIiZK).

Komitet Główny odbył 8 posiedzeń, a Prezydium – 6; protokoły z tych posiedzeń znajdują się w sekretariacie Olimpiady. Dnia 26 stycznia 1996 r. odbyło się jednodniowe seminarium dla członków komitetów okręgowych, poświęcone organizacji zawodów II stopnia w okręgach.

5.2.2. Komitety Okręgowe

Komitet Okręgowy w Warszawie

przewodniczący:

dr Krzysztof Diks, Uniwersytet Warszawski,

członkowie:

mgr Marcin Kubica, Uniwersytet Warszawski,
mgr Adam Malinowski, Uniwersytet Warszawski,
mgr Wojciech Plandowski, Uniwersytet Warszawski,
dr Andrzej Walat, OEIiZK.

Siedzibą Komitetu Okręgowego jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie, ul. Raszyńska 8/10.

Komitet Okręgowy we Wrocławiu

przewodniczący:

prof. dr hab. Maciej M. Sysło, Uniwersytet Wrocławski,

zastępca przewodniczącego:

dr Krzysztof Loryś, Uniwersytet Wrocławski,

sekretarz:

inż. Maria Woźniak, Uniwersytet Wrocławski,

członkowie:

mgr inż. Grażyna Koba, WOM Wrocław,
mgr Jacek Jagiełło, Uniwersytet Wrocławski,
dr Witold Karczewski, Uniwersytet Wrocławski,
mgr Paweł Keller, Uniwersytet Wrocławski.

Siedzibą Komitetu Okręgowego jest Instytut Informatyki Uniwersytetu Wrocławskiego we Wrocławiu, ul. Przesmyckiego 20.

Komitet Okręgowy w Toruniu

przewodniczący:

prof. dr hab. Józef Słomiński, Uniwersytet Mikołaja Kopernika w Toruniu,

zastępca przewodniczącego:

dr Mirosława Skowrońska, Uniwersytet Mikołaja Kopernika w Toruniu,

sekretarz:

dr Bolesław Wojdyło, Uniwersytet Mikołaja Kopernika w Toruniu,

członkowie:

mgr Anna Kwiatkowska, IV Liceum Ogólnokształcące w Toruniu,
dr Krzysztof Skowronek, Kuratorium Oświaty w Toruniu.

Siedzibą Komitetu Okręgowego w Toruniu jest Wydział Matematyki i Informatyki Uniwersytetu Mikołaja Kopernika w Toruniu, ul. Chopina 12/18.

5.3. Jury

Sprawdzaniem zadań konkursowych zajmowało się jury Olimpiady, któremu przewodniczył prof. dr hab. inż. Stanisław Waligórski. W pracach jury brali udział: sekretarz naukowy Olimpiady dr Andrzej Walat oraz pracownicy Instytutu Informatyki UW i studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego:

Jacek Chrząszcz	– student,
mgr Marcin Engel	– doktorant,
Marcin Jurdziński	– student,
Piotr Krysiuk	– student,
Albert Krzymowski	– student,
mgr Marcin Kubica	– doktorant,
Marcin Madey	– student,
Marek Pawlicki	– student,
mgr Piotr F. Sawicki	– pracownik,
Krzysztof Sobusiak	– student,
mgr Krzysztof Stencel	– doktorant,
Tomasz Śmigielski	– student.

5.4. Zawody I stopnia

W zawodach I stopnia III Olimpiady wzięło udział 476 uczniów, którzy nadesłali łącznie 1364 rozwiązania zadań, w tym:

310 rozwiązań zadania ZAMEK,

465 rozwiązań zadania PRĘTY,

277 rozwiązań zadania MOKRA ROBOTA,

312 rozwiązań zadania GOŃCY.

Jury podjęło decyzję o sprawdzeniu 27 rozwiązań przysłanych przez uczniów w kopiach zapasowych:

6 rozwiązań zadania ZAMEK,

7 rozwiązań zadania PRĘTY,

7 rozwiązań zadania MOKRA ROBOTA,

7 rozwiązań zadania GOŃCY.

Z rozwiązaniami

czterech zadań nadeszło	– 186 prac,
trzech zadań	– 114 prac,

dwóch zadań – 102 prace,
jednego zadania – 74 prace.

Przysłano 11 dyskietek, które były częściowo lub całkowicie nieczytelne, z czego z 8 z nich udało się odzyskać większość zapisów za pomocą programów Norton Utility. Nie udało się odzyskać informacji z trzch dyskietek. Na 31 dyskietkach wykryto i usunięto różnego rodzaju wirusy.

Uczestnicy I etapu reprezentowali 48 województw. Nie było uczniów tylko z województwa przemyskiego.

Najliczniej były reprezentowane następujące województwa:

st. warszawskie	– 46 uczniów,	nowosądeckie	– 14,
katowickie	– 43,	bydgoskie	– 13,
gdańskie	– 37,	kieleckie	– 13,
rzeszowskie	– 30,	zielonogórskie	– 12,
krakowskie	– 28,	lubelskie	– 11,
szczecińskie	– 21,	bielskie	– 10,
tarnowskie	– 19,	częstochowskie	– 10,
łódzkie	– 16,	poznańskie	– 10.
wrocławskie	– 16,		

W zawodach I stopnia najliczniej były reprezentowane szkoły:

1. I L.O. im. S. Konarskiego z Mielca – 14 uczniów,
2. III L.O. im. Marynarki Wojennej RP z Gdyni – 12,
3. L.O. im. W. Jagiełły z Dębicy – 9,
4. I L.O. im. S. Staszica z Ostrowca Św. – 8,
5. L.O. im. A. Mickiewicza z Góry – 7,
6. I L.O. im. S. Dubois z Koszalinia – 6,
7. V L.O. im. A. Witkowskiego z Krakowa – 6,
8. II L.O. im. S. Batorego z Warszawy – 6,
9. XXVII L.O. im. T. Czackiego z Warszawy – 6,
10. L.O. im. B. Chrobrego ze Szprotawy – 6,
11. IX L.O. im. Giuseppe di Vittorio z Gdańska – 5,
12. XII L.O. z Gdańska – 5,
13. XXXI L.O. im. L. Zamenhofa z Łodzi – 5,
14. L.O. im. M. Skłodowskiej-Curie z Piły – 5,
15. VIII L.O. im. A. Mickiewicza z Poznania – 5,
16. V L.O. im. J. Poniatowskiego z Warszawy – 5,
17. XIV L.O. ze Szczecina – 5,
18. I L.O. im. M. Kopernika z Gdańska – 4,
19. I L.O. im. S. Żeromskiego z Jeleniej Góry – 4,
20. I L.O. im. Powstańców Śląskich z Rybnika – 4,

21. II L.O. im. Jana III Sobieskiego z Krakowa – 4,
22. IV L.O. im. M. Kopernika z Rzeszowa – 4,
23. XIV L.O. im. S. Staszica z Warszawy – 4,
24. V L.O. im. A. Asnyka ze Szczecina – 4,
25. XIV L.O. im. Polonii Belgijskiej z Wrocławia – 4.

Do zawodów I stopnia decyzją Komitetu Głównego Olimpiady dopuszczono 4 uczniów ze szkół podstawowych: nr 10 ze Stalowej Woli, nr 11 z Jeleniej Góry, nr 11 z Nowego Targu i nr 3 z Zakopanego.

W zawodach I stopnia najliczniej były reprezentowane szkoły z następujących miast:

Warszawa	– 40 uczniów,	Rzeszów	– 10,
Kraków	– 24,	Bydgoszcz	– 9,
Szczecin	– 20,	Poznań	– 9,
Gdańsk	– 17,	Góra	– 8,
Mielec	– 16,	Ostrowiec Św.	– 8,
Gdynia	– 13,	Wrocław	– 8,
Łódź	– 12,	Jelenia Góra	– 7,
Dębica	– 10	Tarnów	– 7.

407 uczniów podało klasę, do której chodzi. W tym:

do I klasy szkoły średniej	– 27 uczniów,
do II klasy szkoły średniej	– 83,
do III klasy szkoły średniej	– 139,
do IV klasy szkoły średniej	– 140,
do V klasy szkoły średniej	– 14,
do VIII klasy szkoły podstawowej	– 4.

Największa liczba uczniów zadeklarowała użycie języka programowania:

Turbo Pascal (Borland)	– 397 uczniów,
Borland C/C++	– 65.

Ponadto posługiwano się językami:

Watcom C/C++	– 5 uczniów,
Basic	– 3,
Clipper	– 1 uczeń,
High Speed Pascal	– 1,
Logo	– 1,
MS Quick Pascal	– 1,
MS Visual Pascal	– 1,
MS Visual Basic	– 1.

Komputerowe wspomaganie umożliwiło sprawdzenie prac z zawodów I stopnia kompletem 44 testów w ciągu kilkunastu godzin.

Komitet Główny podjął decyzję o wykluczeniu z zawodów uczniów oznaczonych następującymi numerami: 133 i 134; 369 i 370; 337, 338, 339, 340, 341 i 342; 230 i 231; 234. Decyzję podjęto po porównaniu kodów nadesłanych rozwiązań, które różniły się jedynie nieistotnymi szczegółami, na przykład:

— nazwami identyfikatorów,
— kolejnością deklarowania procedur,
— kolejnością instrukcji w treści procedur.

Dotyczyło to zadań ZAMEK, PRĘTY i MOKRA ROBOTA u zawodników nr 230 i 231 oraz zadania ZAMEK u zawodnika nr 234, zadania PRĘTY u zawodników nr 133 i 134, zadania GOŃCY u zawodników nr 369 i 370 oraz zadania ZAMEK u zawodników od nr 337 do nr 342.

Sprawdzanie, jak w poprzedniej olimpiadzie, był utrudnione występowaniem takich niedokładności, jak nieprzestrzeganie podanych w treści zadań reguł dotyczących nazywania plików i tworzenia zestawów danych wynikowych – sprawdzanie takich prac trwało znacznie dłużej.

W Tabeli 1 zawarto sumaryczne wyniki z zawodów I stopnia z podziałem na zadania i zakresy uzyskanych punktów.

Tabela 1.

Liczba punktów	ZAMEK		PRĘTY		MOKRA ROBOTA		GOŃCY	
	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo
100	1	0,3%	29	6,2%	17	6,1%	5	1,9%
od 99 do 75	1	0,3%	11	2,4%	18	6,5%	9	2,8%
od 74 do 50	3	1,0%	30	6,5%	27	9,7%	28	9,0%
od 49 do 1	172	55,4%	350	75,3%	198	71,6%	216	69,1%
0	133	43,0%	45	9,6%	17	6,1%	54	17,2%

Wyniki za wszystkie 4 zadania są zawarte w Tabeli 2.

Tabela 2.

SUMA	liczba uczniów	procentowo
400 pkt.	0	0,0%
od 399 do 300 pkt.	5	1,0%
od 299 do 200 pkt.	26	5,5%
od 199 do 1 pkt.	411	86,9%
0 pkt.	31	6,6%

Prezydium Komitetu Głównego, przy udziale jury, rozpatrzyło 9 reklamacji, z których trzy wniosły zmiany do punktacji zawodów I stopnia.

5.5. Zawody II stopnia

Do zawodów II stopnia zakwalifikowano 107 uczniów, którzy osięgnęli w zawodach I stopnia wynik nie mniejszy niż 108 pkt.

Zawody II stopnia odbyły się w dniach 1–3 lutego 1996 roku w trzech okręgach (w Toruniu, Wrocławiu i w Warszawie) i uczestniczyli w nich uczniowie z następujących województw:

w Toruniu – 20 uczniów z województw:

bydgoskiego	– 7,	płockiego	– 1,
gdańskiego	– 10,	toruńskiego	– 1,
koszalińskiego	– 1,		

we Wrocławiu – 40 uczniów z województw:

bielskiego	– 1,	opolskiego	– 2,
częstochowskiego	– 4,	pilskiego	– 1,
jeleniogórskiego	– 1,	piotrkowskiego	– 2,
kaliskiego	– 3,	poznańskiego	– 1,
katowickiego	– 2,	sieradzkiego	– 2,
konińskiego	– 1,	szczecińskiego	– 6,
krakowskiego	– 6,	wałbrzyskiego	– 1,
łódzkiego	– 4,	wrocławskiego	– 3,

w Warszawie – 47 uczniów z województw:

białskopodlaskiego	– 1,	łódzkiego	– 2,
chełmskiego	– 1,	rzeszowskiego	– 12,
ciechanowskiego	– 1,	st. warszawskiego	– 20,
katowickiego	– 2,	suwalskiego	– 1,
kieleckiego	– 2,	tarnowskiego	– 1,

krośnieńskiego – 1, zamojskiego – 1.
lubelskiego – 2,

W zawodach II stopnia najliczniej były reprezentowane szkoły:

1. III L.O. im. Marynarki Wojennej z Gdyni – 6 uczniów,
2. I L.O. im. S. Konarskiego z Mielca – 6,
3. XXXI L.O. im. L. Zamenhofa z Łodzi – 4
4. XXVII L.O. im. T. Czackiego z Warszawy – 4,
5. V L.O. im. A. Witkowskiego z Krakowa – 3,
6. II L.O. im. St. Batorego z Warszawy – 3,
7. V L.O. im. Ks. J. Poniatowskiego z Warszawy – 3,
8. XIV L.O. im. S. Staszica z Warszawy – 3.

Najliczniej były reprezentowane szkoły z następujących miast:

Warszawa	– 20 uczniów,	Szczecin	– 6,
Mielec	– 8,	Bydgoszcz	– 5,
Gdynia	– 6,	Kraków	– 5.
Łódź	– 6,		

1 lutego odbyła się sesja próbna, na której rozwiązywano nie liczące się do ogólnej klasyfikacji zadanie PIĘĆ MONET. W dniach konkursowych zawodnicy rozwiązywali zadania: WIEŻE oceniane maksymalnie na 94 pkt. plus ewentualnie 6 punktów uznaniowych oraz SŁOWA FIBONACCIEGO i HAZARD oceniane każde maksymalnie na 47 pkt. plus ewentualnie 3 punkty uznaniowe.

Sprawdzanie rozwiązań zadań z zawodów II stopnia było również komputerowo wspomagane – zastosowano łącznie 36 testów.

W Tabeli 3 zawarto sumaryczne wyniki z zawodów II stopnia z podziałem na zadania i zakresy uzyskanych punktów.

Tabela 3.

Liczba punktów	WIEŻE		Liczba punktów	SŁOWA FIBONACCIEGO		HAZARD	
	liczba uczniów	procentowo		liczba uczniów	procentowo	liczba uczniów	procentowo
100	3	2,8%	50	1	0,9%	0	0,0%
od 99 do 75	8	7,5%	od 49 do 37	16	15,0%	1	0,9%
od 74 do 50	4	3,7%	od 36 do 25	5	4,7%	0	0,0%
od 49 do 1	83	77,6%	od 24 do 1	78	72,9%	89	83,2%
0	9	8,4%	0	7	6,5%	17	15,9%

Wyniki za wszystkie 3 zadania są zawarte w Tabeli 4.

Tabela 4.

SUMA	liczba uczniów	procentowo
200 pkt.	0	0,0%
od 199 do 150 pkt.	6	5,6%
od 149 do 100 pkt.	7	6,5%
od 99 do 1 pkt.	94	87,9%
0 pkt.	0	0,0%

5.6. Zawody III stopnia

Zawody III stopnia odbyły się w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie w dniach od 25 do 29 marca 1996 roku.

W zawodach III stopnia wzięło udział 38 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 57 pkt., z województw:

st. warszawskie	– 10 uczniów,	częstochowskie	– 1 uczeń,
gdańskie	– 6,	konińskie	– 1,
krakowskie	– 5,	koszalińskie	– 1,
łódzkie	– 3,	poznańskie	– 1,
bydgoskie	– 2,	rzeszowskie	– 1,
kaliskie	– 2,	sieradzkie	– 1,
wrocławskie	– 2,	szczecińskie	– 1,
		wałbrzyskie	– 1.

W zawodach III stopnia najliczniej były reprezentowane szkoły:

1. III L.O. im. Marynarki Wojennej z Gdyni – 4 uczniów,
2. V L.O. im. A. Witkowskiego z Krakowa – 3,
3. XXXI L.O. im. L. Zamenhofa z Łodzi – 3,
4. II L.O. im. St. Batorego z Warszawy – 2,
5. XIV L.O. im. S. Staszica z Warszawy – 2,
6. XXVII L.O. im. T. Czackiego z Warszawy – 2,
7. XIV L.O. im. Polonii Belgijskiej z Wrocławia – 2.

25 marca odbyła się sesja próbna, na której rozwiązywano nie liczące się do ogólnej klasyfikacji zadanie PERMUTACJE SŁÓW. W dniach konkursowych zawodnicy rozwiązywali 4 zadania: AGENCI, KULE, NIE ATAKUJĄCE SIĘ SKOCZKI i WYRÓWNYWANIE SŁÓW, które były oceniane na maksymalną

liczbę 47 punktów z możliwością uzyskania dodatkowych 3 punktów uznaniowych za każde zadanie.

Sprawdzanie rozwiązań zadań z zawodów III stopnia było również komputerowo wspomagane – zastosowano łącznie 57 testów.

W Tabeli 5 zawarto sumaryczne wyniki z zawodów III stopnia z podziałem na zadania i zakresy uzyskanych punktów.

Tabela 5.

Liczba punktów	AGENCI		KULE		NIE ATAKUJĄCE SIĘ SKOCZKI		WYRÓWNYWANIE SŁÓW	
	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo
50	0	0,0%	0	0,0%	0	0,0%	0	0,0%
od 49 do 37	2	5,3%	1	2,6%	7	18,4%	3	7,9%
od 36 do 25	8	21,1%	0	0,0%	0	0,0%	4	10,5%
od 24 do 1	27	71,0%	36	94,8%	23	60,5%	29	76,3%
0	1	2,6%	1	2,6%	8	21,1%	2	5,3%

Wyniki za wszystkie 4 zadania są zawarte w Tabeli 6.

Tabela 6.

SUMA	liczba uczniów	procentowo
200 pkt.	0	0,0%
150 pkt. – max.	1	2,6%
od 149 do 100 pkt.	2	5,3%
od 99 do 1 pkt.	35	92,1%
0 pkt.	0	0,0%

W dniu 29 marca 1996 roku w Auli Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych w Warszawie odbyło się zakończenie III Olimpiady Informatycznej 1995/96, w trakcie którego ogłoszono wyniki finału Olimpiady i rozdano nagrody ufundowane przez: Ministerstwo Edukacji Narodowej oraz firmy: OFEK, OPTIMUS SA, Micrografx, Microsoft, Wydawnictwo Naukowe PWN i Wydawnictwa Szkolne i Pedagogiczne.

Tabela 7 zawiera listę laureatów III Olimpiady Informatycznej.

Tabela 7.

L.p.	Imię i nazwisko ucznia szkoła	Nagroda i jej fundator
Laureaci I nagrody		
1	Andrzej Gąsienica-Samek XIV L.O. im. S. Staszica, Warszawa	Komputer PC 486DX4 – OFEK
2	Piotr Zieliński VIII L.O. im. A. Mickiewicza, Poznań	Komputer PC 486DX2 – MEN
3	Jakub Pawlewicz III L.O. im. Marynarki Wojennej RP, Gdynia	Drukarka atramentowa, Office i Micrografx Suite – Optimus, Microsoft i Micrografx
Laureaci II nagrody		
4	Adam Borowski I L.O. im. M. Skłodowskiej-Curie, Starogard Gdański	Drukarka atramentowa – OFEK
5	Eryk Kopczyński XIV L.O. im. S. Staszica, Warszawa	Drukarka atramentowa – MEN
6	Marcin Sawicki II L.O. im. St. Batorego, Warszawa	Office, FoxPro, FlowChar 3.0 – Microsoft, Micrografx
7	Tomasz Waleń L.O. im. T. Kościuszki, Ziębice	Office, FoxPro, FlowChar 3.0 – Microsoft, Micrografx
Laureaci III nagrody		
8	Wojciech Puchar III L.O. im. Marynarki Wojennej RP, Gdynia	VBasic, FlowChar 2.0 – Microsoft, Micrografx
9	Przemysław Gordinowicz XXXI L.O. im. L. Zamenhofa, Łódź	VC++, FlowChar 2.0 – Microsoft, Micrografx
10	Mirosław Oksiucik III L.O. im. Marynarki Wojennej RP, Gdynia	VC++, FlowChar 2.0 – Microsoft, Micrografx
11	Witold Jarnicki V L.O. im. A. Witkowskiego, Kraków	VC++, FlowChar 2.0 – Microsoft, Micrografx

Wszyscy finaliści otrzymali książki ufundowane przez Wydawnictwo Naukowe PWN oraz drobne upominki ufundowane m.in. przez firmę Peryt.

Ogłoszono również komunikat o powołaniu reprezentacji Polski na VIII Międzynarodową Olimpiadę Informatyczną do Veszprém na Węgrzech w składzie:

Andrzej Gąsienica-Samek,
Piotr Zieliński,
Jakub Pawlewicz,
Adam Borowski;
oraz rezerwowi: Eryk Kopczyński i Marcin Sawicki.

Uczniowie wyróżnieni w zawodach II stopnia i zakwalifikowani do zawodów III stopnia Olimpiady mogą być zwolnieni z egzaminu dojrzałości z przedmiotu informatyka na mocy § 9 ust.1 pkt 2 Zarządzenia Ministra Edukacji Narodowej z dnia 14 września 1992 r. Sekretariat olimpiady wystawił łącznie 38 zaświadczeń o zakwalifikowaniu do zawodów III stopnia celem przedłożenia dyrekcji szkoły. Laureaci i finaliści mogą być zwolnieni z egzaminów wstępnych do wielu szkół wyższych na mocy uchwał senatów uczelni podjętych na wniosek MEN zgodnie z art. 141 ust. 1 ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65, poz. 385). Sekretariat wystawił łącznie 11 zaświadczeń o uzyskaniu tytułu laureata i 27 zaświadczeń o uzyskaniu tytułu finalisty III Olimpiady Informatycznej celem przedłożenia władzom szkół wyższych.

Komitet Główny przyznał również nagrody pieniężne następującym nauczycielom za ich wkład w przygotowanie finalistów Olimpiady (kolejność alfabetyczna):

1. mgr Alina Gościniak
   VIII L.O. im. A. Mickiewicza
   w Poznaniu
   – nauczycielka laureata Piotra Zielińskiego.
2. dr Krzysztof Loryś
   Instytut Informatyki
   Uniwersytetu Wrocławskiego
   – nauczyciel laureta Tomasza Walenia oraz finalistów: Stanisława Paśki, Jacka Suligi.
3. mgr Dzierżysław Malina
   V L.O. im. A. Witkowskiego
   w Krakowie
   – nauczyciel laureata Witolda Jarnickiego oraz finalistów: Adama Rudnickiego, Michała Ślusarczyka.
4. mgr Jarosław Milski
   XXXI L.O. im. L. Zamenhofa
   w Łodzi
   – nauczyciel laureata Przemysława Gordinowicza.
5. mgr inż. Małgorzata Rostkowska
   XIV L.O. im. S. Staszica
   w Warszawie
   – nauczycielka laureata Andrzeja Gąsienicy-Samka.
6. mgr inż. Małgorzata Szatybełko
   I L.O. im. M. Skłodowskiej-Curie
   w Starogardzie Gdańskim
   – nauczycielka laureata Adama Borowskiego.

7. mgr Joanna Śmierzchalska – nauczycielka laureatów: Jakuba Pawlewicza, Mirosława Oksiucika, Wojciecha Puchara oraz finalisty Marcina Stefaniaka.
   III L.O. im. Marynarki Wojennej RP w Gdyni
8. mgr inż. Elżbieta Urbaniak – nauczycielka finalistów: Michała Łukasika, Arkadiusza Talmy.
   XXXI L.O. im. L. Zamenhofa w Łodzi
9. mgr Rafał Wysocki – nauczyciel finalistów: Przemysława Jaroszewskiego, Krzysztofa Majewskiego.
   XXVII L.O. im. T. Czackiego w Warszawie

W dniach 22–24 lipca br. zorganizowano w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie seminarium dla reprezentacji Polski oraz zawodników rezerwowych na Olimpiadę Międzynarodową. W zajęciach, których program opracował dr Krzysztof Diks, wzięli udział: Andrzej Gąsienica-Samek, Piotr Zieliński, Jakub Pawlewicz, Adam Borowski, Eryk Kopczyński, Marcin Sawicki i Tomasz Waleń.

6. VIII MIĘDZYNARODOWA OLIMPIADA INFORMATYCZNA

6.1. Przebieg i organizacja

Ósma Międzynarodowa Olimpiada Informatyczna IOI'96 (*International Olympiad in Informatics*) odbyła się w Veszprém na Węgrzech w dniach od 25 lipca do 1 sierpnia 1996 roku. Polskę reprezentowali laureaci III krajowej Olimpiady Informatycznej: Andrzej Gąsienica-Samek, Piotr Zieliński, Jakub Pawlewicz i Adam Borowski. Opiekunami polskiej ekipy byli: prof. dr hab. inż. Stanisław Waligórski (kierownik ekipy), dr Krzysztof Diks (zastępca kierownika) i dr Piotr Chrząstowski-Wachtel. Wysłanie tak licznej ekipy towarzyszącej było spowodowane chęcią zebrania doświadczeń przed planowaną na przyszły rok w Polsce Olimpiadą Informatyczną Centralnej Europy.

Program tej olimpiady był podobny do programu olimpiad w latach poprzednich i ze względu na tę jego typowość warto go przytoczyć w całości:

Czwartek 25 lipca 1996 – przyjazd.

Dzień 1. Ceremonia otwarcia, wycieczka po Veszprém, zapoznanie się zawodników z komputerami i konfiguracją oprogramowania.

Dzień 2. Pierwszy dzień zawodów (27 lipca).

Dzien 3. Wycieczka.

Dzień 4. Drugi dzień zawodów (29 lipca).

Dzień 5. Wycieczka.

Dzień 6. Ceremonia ogłoszenia wyników i wręczenia nagród. Zamknięcie Olimpiady.

Czwartek 1 sierpnia 1996 – odjazd.

IX Międzynarodowa Olimpiada Informatyczna odbędzie się w Cape Town w Republice Południowej Afryki w dniach od 30 listopada do 7 grudnia 1997 roku i jej wstępny program wygląda podobnie.

Pierwszy raz wprowadzono zatwierdzanie tekstów zadań przez Zgromadzenie Ogólne Olimpiady (*General Assembly*, dawniej zwane Jury Międzynarodowym – *International Jury*, w skład którego wchodzą opiekunowie wszystkich ekip) i tłumaczenie ich na języki zawodników późnym wieczorem dnia poprzedzającego zawody. Zgromadzenie zostaje odizolowane od zawodników od początku tego

wieczornego spotkania aż do końca zawodów w dniu następnym. Dotychczas te spotkania Zgromadzenia zaczynały się o 5 rano w dniu zawodów, ale ze względu na dużą liczbę zadań do rozważenia i przetłumaczenia – po trzy w każdym dniu zawodów – wprowadzenie tej innowacji było konieczne.

Wśród zadań olimpiady międzynarodowej zagościły już na trwałe, jak się wydaje, zadania wymagające napisania programu konwersacyjnego, współpracującego z pewnymi modułami przygotowanymi przez organizatorów. W tej olimpiadzie było to zadanie GRA. Pozostałe zadania wymagały napisania programów działających w trybie wsadowym. Nie było, w przeciwieństwie do poprzedniej olimpiady, żadnego zadania, które miało być rozwiązane bez pomocy komputera.

Rozwiązania zadań były oceniane wyłącznie za pomocą komputera, zaś używane do tego oprogramowanie było w zasadzie podobne do tego, które stosowano w poprzednim roku, choć było widać pewne istotne usprawnienia procedury. Dzięki temu ocenianie przebiegało szybko i sprawnie. Nie uniknięto jednak pewnych pomyłek w danych testowych, wskutek czego część postępowania oceniającego trzeba było powtórzyć. Regulamin oceniania pozostał taki sam, jak w ubiegłym roku, zobacz p. 6.2.2 w [2].

6.2. Wyniki

Sklasyfikowano 215 zawodników z 57 krajów. Do zdobycia było w sumie 200 punktów, po 100 punktów za trzy zadania w każdym dniu zawodów.

Polscy zawodnicy uzyskali (kolejno podajemy: miejsce na liście wszystkich zawodników według liczby zdobytych punktów, liczbę punktów, medal):

Piotr Zieliński	3	191	złoty
Jakub Pawlewicz	14	180	złoty
Andrzej Gąsienica-Samek	27	175	srebrny
Adam Borowski	68	136	brązowy

Piotr Zieliński i Jakub Pawlewicz brali udział w poprzednich Olimpiadach Międzynarodowych, uzyskując srebrne medale w 1995 roku w Eindhoven. Zdobyli również srebrne medale w Olimpiadzie Informatycznej Centralnej Europy CEOI'95 w Szeged. Jakub Pawlewicz zdobył też brązowe medale w CEOI'94 w Cluj i w Międzynarodowej Olimpiadzie 1994 roku w Sztokholmie.

Nasi dwaj pozostali zawodnicy byli w zawodach międzynarodowych nowicjuszami. W zeszłym roku Jakub Pawlewicz był pierwszy na liście srebrnych medalistów IOI'95. W zbliżonej sytuacji znalazł się teraz Andrzej Gąsienica-Samek – był pierwszy na liście srebrnych medalistów.

Największe sukcesy odnieśli czterej zawodnicy Chin – każdy z nich zdobył złoty medal. Zawodnicy Rosji zdobyli trzy złote medale i jeden srebrny. Czterej zawodnicy słowaccy zdobyli dwa złote medale i dwa srebrne. Pozostałe kraje zdobyły po co najwyżej jednym złotym medalu, których w sumie było 20.

Puchar IFIPu, ufundowany przez Komitet do spraw Kształcenia Międzynarodowej Federacji Przetwarzania Informacji (*International Federation for Information Processing*), nagrodę przechodnią przeznaczoną dla najlepszego zawodnika, zdobył w tym roku Daniel Kral z Czech, uzyskując jako jedyny 196 punktów.

Pomiędzy Danielem Kralem i Piotrem Zielińskim był tylko Wang Xiaochuan z Chin, zdobywca 194 punktów.

Po zsumowaniu zdobytych punktów przez wszystkich zawodników danego kraju, na pierwszym miejscu znalazły się Chiny z 736 punktami, a potem kolejno: Rosja - 709, Słowacja – 684, Polska – 682, Rumunia – 652, Tajwan – 647, Litwa – 611, Iran – 607.

Warto na koniec dodać, że w przeprowadzonych wyborach do Międzynarodowego Komitetu Olimpiady na trzy zwalniane miejsca zostały wybrane: USA, Polska i Iran. Będziemy więc mieli swojego reprezentanta we władzach olimpiady.

7. TEKSTY I ROZWIĄZANIA ZADAŃ III OLIMPIADY INFORMATYCZNEJ

W tym rozdziale zamieszczamy teksty zadań III Olimpiady Informatycznej oraz omówienie ich rozwiązań. Każdemu zadaniu jest poświęcony osobny punkt, który składa się z następujących części: treści zadania, opisu rozwiązania (lub rozwiązań), omówienia rozwiązań podanych przez uczniów oraz krótkiej charakterystyki testów. Autorami poszczególnych punktów są autorzy zadań.

Teksty zadań zawierają na ogół opisowe definicje pojęć użytych w treści zadania, sformułowanie samego zadania, opis i przykłady postaci danych wejściowych i wyjściowych dla tego zadania oraz uwagi o nazwach i postaci plików z rozwiązaniami. Teksty zadań w zawodach I stopnia są rozsyłane do wszystkich kuratoriów i szkół średnich w kraju. Teksty zadań w zawodach II i III stopnia są wręczane uczniom, którzy się do nich zakwalifikowali, w chwili rozpoczęcia zawodów. W tym przypadku, przez pierwsze pół godziny, w razie pojawienia się wątpliwości, uczniowie mogą kierować na piśmie do członków Komitetu Głównego i Jury pytania, na które jest udzielana jedynie jedna z trzech odpowiedzi: TAK, NIE lub BEZ ODPOWIEDZI. W kilku przypadkach pytania uczniów spowodowały uściślenie treści zadań przez Komitet i ogłoszenie uzupełnienia wszystkim uczniom biorącym udział w zawodach II lub III stopnia.

Rozwiązania zadań podane przez autorów zadań zawierają najczęściej opis metody, która z jednej strony jest dość prosta, a z drugiej – należy do najlepszych rozwiązań tego zadania. W tych punktach starano się uczynić kompromis między złożonością opisu a jakością rozwiązania. Podana metoda rozwiązania jest zawsze uzasadniona, a czasem zawiera podstawy teoretyczne, na których jest oparta. Integralną częścią rozwiązań są kompletne programy w języku Pascal realizujące te rozwiązania, umieszczone na dyskietce załączonej do tego opracowania. Pliki ze źródłowymi tekstami programów mają nazwy podane w treści zadań. Każde zadanie ma osobną kartotekę na dyskietce, nazwaną skrótem nazwy zadania, w której oprócz tekstu programu znajduje się plik ze spakowanymi testami używanymi przez Jury do sprawdzania rozwiązań uczniowskich.

Omówienia rozwiązań podanych przez uczniów zostały sporządzone przez autorów zadań po przeglądnięciu reprezentatywnej liczby prac nadesłanych w zawodach I stopnia oraz wszystkich prac w przypadku zadań z zawodów II i III stopnia. W tym omówieniu starano się krótko opisać inne metody rozwiązywania podane przez uczniów, które efektywnością nie odbiegają od rozwiązań podanych przez autorów zadań. Znaczną część tych omówień wypełniają uwagi o błędach popełnionych przez uczniów na różnych etapach rozwiązywania zadania.

Na końcu punktu poświęconego jednemu zadaniu zawarto omówienie testów (tylko niektóre z nich są w pełnej postaci), których Jury używało do sprawdzania poprawności rozwiązań oraz do określenia punktacji rozwiązań. Wszystkie testy są umieszczone na dyskietce załączonej do tego opracowania.

7.1. Zawody I stopnia

7.1.1. Zadanie ZAMEK (Autor: Krzysztof Diks)

Treść zadania ZAMEK

Poszukiwacze skarbów zdobyli mapę zamku, w którego podziemiach znajduje się olbrzymi skarb. Mapa jest naniesiona na siatkę kwadratową, której węzły mają współrzędne całkowite. Lewy dolny róg (węzeł) siatki ma współrzędne (0,0), a przeciwległy, prawy górny róg współrzędne (10000, 10000). Na mapie zamek ma kształt wielokąta, którego boki leżą na liniach siatki. Kolejne dwa boki wielokąta są zawsze prostopadłe. Brzeg tego wielokąta jest łamaną zamkniętą zwykłą, tzn. każdy jej wierzchołek należy do dokładnie dwóch odcinków łamanej, a każdy inny punkt – do jednego. Odcinki linii tworzących siatkę zawarte w wielokącie, w tym także każdy bok wielokąta, przedstawiają odcinki podziemnych korytarzy zamku. W jednym z węzłów (tzn. na przecięciu linii siatki) należącym do wielokąta jest zaznaczony punkt wejścia do podziemi, a w innym węźle, również należącym do wielokąta – punkt, w którym jest ukryty skarb.

Chcemy obliczyć długość najkrótszej drogi podziemnymi korytarzami zamku od punktu wejścia do podziemi do punktu, w którym ukryty jest skarb. Przyjmujemy, że jednostką długości jest długość boku pojedynczej kratki na siatce.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego ZAM.IN następujące dane o zamku:
  - liczbę wierzchołków n wielokąta przedstawiającego zamek, $4 \leq n \leq 5000$,

  - współrzędne wierzchołków w kolejności ich występowania przy obchodzeniu wielokąta,
  - współrzędne punktu wejścia do podziemi i punktu, w którym ukryty jest skarb;
- znajduje długość najkrótszej drogi podziemnymi korytarzami zamku (po liniach siatki), od punktu wejścia do podziemi do punktu, w którym znajduje się skarb;
- zapisuje wynik w pliku tekstowym ZAM.OUT.

Wejście

W pierwszym wierszu pliku tekstowego ZAM.IN jest zapisana jedna liczba całkowita n z zakresu [4..5000] – jest to liczba wierzchołków wielokąta przedstawiającego zamek.

W każdym z kolejnych n wierszy są zapisane dwie liczby całkowite z zakresu [0..10000] oddzielone pojedynczym odstępem – są to współrzędne kolejnego wierzchołka wielokąta.

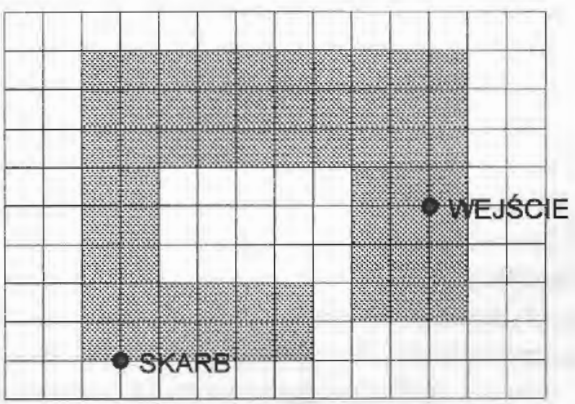
Następnie, w przedostatnim wierszu pliku są zapisane w takim samym formacie współrzędne punktu wejścia do podziemi, a w ostatnim wierszu – współrzędne punktu położenia skarbu.

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego ZAM.OUT należy zapisać jedną liczbę całkowitą – długość najkrótszej drogi od punktu wejścia do podziemi do punktu w którym ukryty jest skarb.

Przykład

Opisem zamku przedstawionego na rysunku jest następujący plik tekstowy ZAM.IN



```
10
9 6
9 2
12 2
12 9
2 9
2 1
8 1
8 3
4 3
4 6
11 5
3 1
```

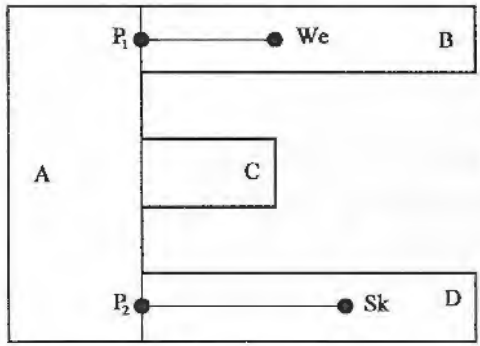
Poprawnym rozwiązaniem jest w tym przypadku następujący plik ZAM.OUT:

```
14
```

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

Rozwiązanie zadania ZAMEK

W naszych rozważaniach punkt zawsze oznacza węzeł siatki. Pisząc wielokąt mamy zawsze na myśli wielokąt o bokach równoległych do boków siatki i wierzchołkach w jej węzłach. Jeżeli P jest punktem, to jego współrzędne oznaczamy przez $x(P)$ i $y(P)$. Mówimy, że punkt P (symetrycznie Q) **leży na lewo** (**na prawo**) od punktu Q (punktu P) jeżeli $x(P) < x(Q)$ ($x(P) > x(Q)$). Podobnie, $P(Q)$ leży **poniżej** Q (**powyżej**) P jeśli $y(P) < y(Q)$ ($y(P) > y(Q)$). Te definicje można łatwo rozszerzyć na figury składające się z wielu punktów. Powiemy, że figura F leży na lewo od figury G, jeżeli każdy punkt z F leży na lewo od każdego punktu z G. Podobnie definiujemy położenie jednej figury względem drugiej na prawo, poniżej i powyżej. **Drogą** między punktami P i Q nazywamy łamaną zwykłą o początku w P i końcu w Q, której dwa kolejne odcinki leżą na liniach siatki i są do siebie prostopadłe. **Długość** drogi definiujemy jako sumę długości jej składowych odcinków. Dla punktów P i Q należących do wielokąta W, przez $d_W(P, Q)$ oznaczamy długość najkrótszej drogi łączącej P z Q i całkowicie zawartej w W (brzeg zaliczamy do wielokąta). Wielkość $d_W(P, Q)$ nazywamy **odległością** między punktami P i Q w wielokącie W. Drogę łączącą P i Q w wielokącie W o długości $d_W(P, Q)$ nazywamy **drogą optymalną**. Jeżeli W jest

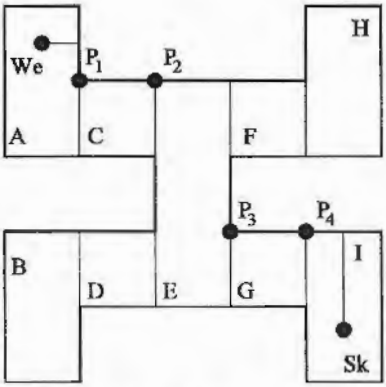


Rysunek 1. Wielokąt w kształcie litery E.

wielokątem-zamkiem, a *We* i *Sk* są punktami, odpowiednio, wejścia do zamku i ukrycia skarbu, to naszym celem jest obliczenie $d_W(We, Sk)$.

Zanim przystąpimy do przedstawienia rozwiązania zadania rozważmy 3 różne wielokąty, dla których odległość między punktem wejścia i punktem ukrycia skarbu będzie łatwo obliczyć, a sposób, w jaki to zrobimy, można będzie zastosować do dowolnego wielokąta.

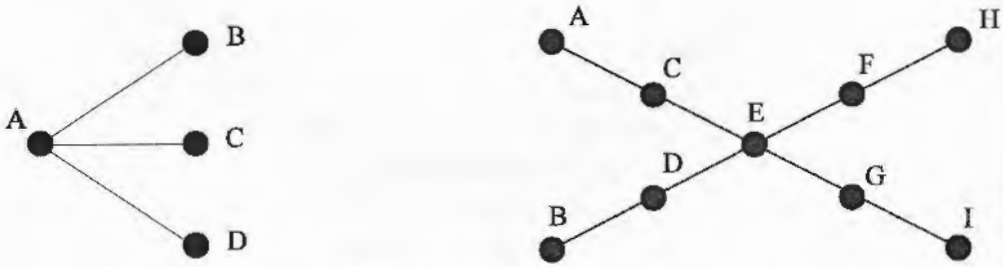
1. Jeśli wielokąt W jest prostokątem, to $d_W(We, Sk) = |x(We) - x(Sk)| + |y(We) - y(Sk)|$, gdzie $|a|$ oznacza wartość bezwzględną liczby a.
2. Rozważmy teraz wielokąt W w kształcie litery E (zob. rys. 1). W tym wypadku W można podzielić na cztery prostokąty oznaczone na rysunku jako A, B, C i D. Jeśli oba punkty *We* i *Sk* należą do tego samego prostokąta lub jeden z nich należy do A, a drugi do jednego z pozostałych trzech prostokątów, to odległość między punktem wejścia i punktem ukrycia skarbu można obliczyć tak samo jak w przypadku 1. Rozważmy przypadek przedstawiony na rys. 1, gdzie *We* należy do prostokąta B, a *Sk* należy do prostokąta D. Zauważmy, że każda optymalna droga z punktu wejścia do pnnktu ukrycia skarbu prowadzi kolejno przez prostokąty B, A i D. Aby obliczyć jej długość wystarczy znaleźć dwa punktu P_1 i P_2 takie, że P_1 jest najbliższym punktem dla *We* leżącym na wspólnym brzegu prostokątów B i A, natomiast P_2 jest najbliższym punktem dla P_1 leżącym na wspólnym brzegu A i D. Łatwo zauważyć, że $d_W(We, Sk) = d_B(We, P_1) + d_A(P_1, P_2) + d_D(P_2, Sk)$.
3. Rozważmy wielokąt W z rys. 2 z zaznaczonymi punktami *We* i *Sk*. Wielokąt W można podzielić na 9 prostokątów $A, B, \ldots, I$. Punkt wejściowy *We* znajduje się w prostokącie A, natomiat skarb znajduje się w prostokącie I. Niech $We, P_1, \ldots, P_4, Sk$ będzie ciągiem punktów zdefiniowanym następująco: P_1 jest punktem na wspólnym brzegu prostokątów A i C położonym najbliżej



Rysunek 2. Trzeci wielokąt.

We, P_2 jest punktem na wspólnym brzegu prostokątów C i E położonym najbliżej P_1, P_3 jest punktem na wspólnym brzegu E i G położonym najbliżej P_2, a P_4 jest najmniej odległym punktem dla P_3 położonym na wspólnym brzegu prostokątów G oraz I. Wówczas $d_W(We, Sk) = d_A(We, P_1) + d_C(P_1, P_2) + d_E(P_2, P_3) + d_G(P_3, P_4) + d_I(P_4, Sk)$. Zauważmy, że każda optymalna droga z *We* do *Sk* prowadzi kolejno przez prostokąty A, C, E, G, I.

Czym się charakteryzują wszystkie podziały wielokątów na prostokąty z punktów 1–3? (W punkcie 1 podział składa się tylko z jednego prostokąta – samego wielokąta W.) Niech *Podz* będzie **podziałem** wielokąta W na prostokąty (zbiorem prostokątów). Powiemy, że dwa prostokąty X, Y są **sąsiednie** w tym podziale, jeśli mają wspólne punkty brzegowe. **Grafem prostokątów** dla danego podziału *Podz* nazywamy graf $G(Podz)$, w którym zbiorem wierzchołków jest zbiór prostokątów podziału. Dwa prostokąty są połączone krawędzią w grafie $G(Podz)$ wtedy i tylko wtedy, gdy są sąsiednie w *Podz*. Zauważmy, że dla każdego z podziałów 1–3 odpowiadający mu graf prostokątów jest drzewem (zob. rys. 3), tzn. spójnym grafem bez cykli.



Rysunek 3. Grafy dla podziałów z punktów 2 i 3.

W drzewie między każdą parą wierzchołków istnieje dokładnie jedna droga. (W grafie **droga** jest ciągiem wierzchołków, z których każde dwa kolejne tworzą w grafie krawędź.) W szczególności, jedyna droga w drzewie podziału, łącząca prostokąt zawierający punkt wejścia do zamku z prostokątem zawierającym punkt ukrycia skarbu, wyznacza ciąg prostokątów przez które prowadzi optymalna droga z punktu wejścia do punktu ukrycia skarbu. Niech *We* i *Sk* będą dowolnymi punktami w wielokącie-zamku *W* i niech *Pr*(*We*) będzie prostokątem w podziale *Podz* wielokąta *W* zawierającym *We*, a *Pr*(*Sk*) prostokątem w tym podziale zawierającym *Sk*. Jeżeli punkt *We* (podobnie *Sk*) leży na wspólnym brzegu dwóch prostokątów, to za *Pr*(*We*) (podobnie za *Pr*(*Sk*)) hierzemy ten prostokąt, dla którego punkt *We* (*Sk*) leży na prawym boku. Rozważmy jedyną drogę w drzewie podziału *G*(*Podz*) łączącą *Pr*(*We*) z *Pr*(*Sk*). Niech $A_1 = Pr(We), A_2, \ldots, A_k = Pr(Sk)$ będą kolejnyuni prostokątami na tej drodze, a $P_0, P_1, \ldots, P_{k-1}, P_k$ ciągiem punktów takich, że $P_0 = We$, P_1 jest punktem na wspólnym brzegu prostokątów A_1 i A_2 najmniej odległym od P_0, P_2 jest punktem na wspólnym brzegu prostokątów A_2 i A_3 najmniej odległym od $P_1, \ldots, P_{k-1}$ jest punktem na wspólnym brzegu prostokątów A_{k-1} i A_k najmniej odległym od P_{k-2}, a $P_k = Sk$. Wówczas

$$d_W(We, Sk) = d_{A_1}(We, P_1) + d_{A_2}(P_1, P_2) + \ldots + d_{A_{k-1}}(P_{k-2}, P_{k-1}) + \\ + d_{A_k}(P_{k-1}, Sk). \qquad (1)$$

Powstaje pytanie, czy każdy wielokąt można podzielić w taki sposób, aby graf prostokątów był drzewem. Odpowiedź jest pozytywna. Podziałów o powyższej własności może być oczywiście wiele. Tutaj pokażemy w jaki sposób skonstruować jeden z nich.

Niech *W* będzie *n*-wierzchołkowym wielokątem. Podzielimy *W* podobnie jak w punktach 1–3. Dla każdego pionowego boku *b*, niech *S*(*b*) będzie najdłuższym odcinkiem pionowym zawierającym *b* i całkowicie zawartym w *W* (pamiętajmy, że boki należą do *W*). Zauważmy, że jeżeli odcinki *S*(*b*) i *S*(*c*) mają punkty wspólne, to się pokrywają (*S*(*b*) = *S*(*c*)). Odcinki *S*(*b*), dla pionowych boków wielokąta *W*, wyznaczają podział *Podz*(*W*) tego wielokąta na prostokąty o rozłącznych wnętrzach. Bok poziomy każdego prostokąta *A* w tym podziale jest całkowicie zawarty w pewnym poziomym boku wielokąta, natomiast bok pionowy – w pewnym odcinku *S*(*b*). Ponieważ poziome boki każdego prostokąta w podziale są zawarte w poziomych bokach wielokąta *W*, graf prostokątów jest drzewem. Podział otrzymany w ten sposób nazywamy **podziałem podłużnym**.

Dotychczasowa dyskusja prowadzi nas do sformułowania następującego algorytmu obliczania długości optymalnej drogi między punktami *We* i *Sk* w danym wielokącie *W*.

Algorytm *DrogaPoSkarb*(*W*, *We*, *Sk*);

1. Oblicz podział podłużny *Podz* wielokąta *W*.
2. Zbuduj drzewo prostokątów *T* dla podziału z kroku 1.
3. Wyznacz prostokąty *Pr*(*We*) i *Pr*(*Sk*) w podziale *Podz*.
4. W drzewie *T* znajdź jedyną drogę $A_1 = Pr(We), A_2, \ldots, A_k = Pr(Sk)$ łączącą *Pr*(*We*) z *Pr*(*Sk*).
5. Korzystając z drogi $A_1, A_2, \ldots, A_k$ oblicz długość najkrótszej drogi łączącej *We* z *Sk* zgodnie ze wzorem (1).

Opiszemy teraz szczegółowo realizację poszczególnych kroków algorytmu.

1. Oblicz podział podłużny *Podz* wielokąta *W*.

Ten krok jest najtrudniejszy w całym algorytmie. Naszym celem jest znalezienie wszystkich prostokątów w podłużnym podziale danego n-kąta *W*. Każdy taki prostokąt wyznaczymy podając współrzędne jego dolnego lewego wierzchołka oraz jego wysokość i szerokość.

Do dalszych rozważań potrzebujemy kilku nowych definicji. Niech $\propto$ będzie porządkiem na pionowych bokach wielokąta *W*, zdefiniowanym następująco: $b_1 \propto b_2$ wtedy i tylko wtedy, gdy albo b_1 i b_2 leżą na różnych, pionowych liniach siatki i linia zawierająca b_1 leży na lewo od linii zawierającej b_2, albo b_1 i b_2 leżą na tej samej linii siatki i bok b_1 znajduje się poniżej boku b_2.

Podobnie definiujemy porządek $\propto$ na bokach poziomych. Dla poziomych boków b_1, b_2, $b_1 \propto b_2$ wtedy i tylko wtedy, gdy albo b_1 i b_2 leżą na różnych, poziomych liniach siatki i linia zawierająca b_1 leży poniżej linii zawierającej b_2, albo b_1 i b_2 leżą na tej samej linii siatki i bok b_1 znajduje się na lewo od boku b_2.

Powiemy, że pionowy bok b wielokąta *W* jest **otwierający** (**zamykający**), jeśli wnętrze wielokąta *W* leży po jego prawej (lewej) stronie. Podobnie klasyfikujemy boki poziome. Poziomy bok b wielokąta *W* jest **otwierający** (**zamykający**), jeśli wnętrze wielokąta *W* leży nad (pod) b. O pionowym boku b powiemy, że **dotyka** prostokąt *A* z podziału **z lewej strony**, jeżeli co najmniej jeden z końców b leży na lewym boku *A*. Podobnie, bok b **dotyka** prostokąt *A* **z prawej strony**, jeżeli co najmniej jeden z końców b leży na prawym boku *A*. Każdy bok pionowy może dotykać co najwyżej dwa różne prostokąty z jednej strony, ale nie więcej niż trzy jednocześnie z obu stron. Niech *A* będzie prostokątem w podziale. **Bokiem otwierającym** dla *A* nazywamy pierwszy pionowy bok w porządku $\propto$ dotykający *A* z lewej strony. O boku tym mówimy, że **otwiera** *A*.

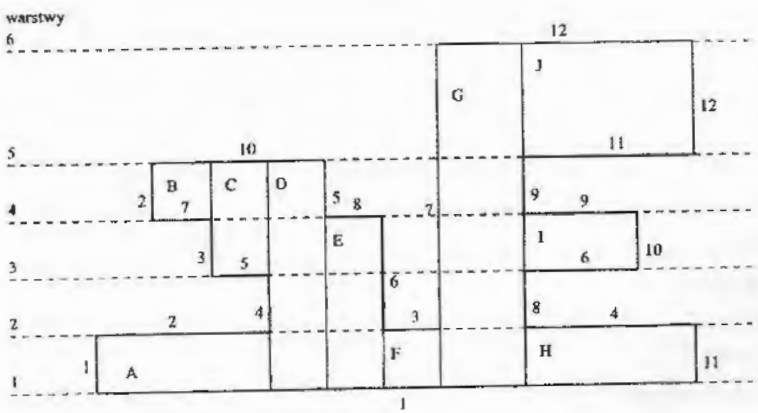
Natomiast **bokiem zamykającym dla** A nazywamy pierwszy bok w porządku $\propto$ dotykający A z prawej strony. O boku tym mówimy, że **zamyka** A.

Niech b będzie bokiem pionowym, a c bokiem poziomym nie mającym wspólnego końca z b. Powiemy, że c jest **aktywny** względem b jeśli istnieje prostokąt A w podziale podłużnym wielokąta W, którego jeden z boków poziomych jest zawarty w c i dla którego bokiem otwierającym jest bok $b' \propto b$, a bokiem zamykającym bok b lub bok b'' taki, że $b \propto b''$. Bok c mający wspólny koniec z b jest **aktywny**, jeśli b zamyka prostokąt o jednym z boków poziomych zawartych w c.

Przekrojem wielokąta W dla pionowego boku b nazywamy ciąg aktywnych, poziomych boków względem b, uporządkowany kolejno od boku położonego najniżej do boku położonego najwyżej.

Poziome linie siatki zawierające boki wielokąta W nazywamy **warstwami**. Warstw może być co najwyżej $n/2$. Ponumerujmy warstwy wielokąta W kolejnymi liczbami 1, 2, ... od warstwy położonej najniżej do warstwy położonej najwyżej. Zauważmy, że w każdym przekroju może znajdować się co najwyżej jeden bok poziomy z każdej warstwy. Dla danego przekroju i poziomego boku a przez $Nast(a)$ oznaczamy poziomy bok w przekroju, który jest położony w najniższej warstwie powyżej warstwy zawierającej a. Jeżeli żadna z warstw powyżej warstwy zawierającej a nie zawiera boku z tego przekroju, to $Nast(a)$ jest nieokreślone. Podobnie, przez $Poprz(a)$ oznaczamy poziomy bok w przekroju, który jest położony w najwyższej warstwie poniżej warstwy zawierającej a. Jeżeli żadna z warstw poniżej warstwy zawierającej a nie zawiera boku z tego przekroju, to $Poprz(a)$ jest nieokreślone.

Przykład. Rozważmy wielokąt W z rysunku 4. Wielokąt W jest podzielony podłużnie. Boki pionowe i boki poziome są ponumerowane zgodnie z porządkiem



Rysunek 4. Wielokąt W z bokami pionowymi (poziomymi) ponumerowanymi zgodnie z $\propto$. Prostokąty $A, B, \ldots, J$ są poetykietowane zgodnie z kolejnością otwierania.

$\propto$. Warstwy są zaznaczone liniami przerywanymi. Bokami otwierającymi (zamykającymi) dla poszczególnych prostokątów są: A–1 (4), B–2 (3), C–3 (4), D–4 (5), E–5 (6), F–6 (7), G–7 (8), H–8 (11), I–8 (10), J–9 (12). Przekrojami dla boków pionowych są następujące ciągi odcinków poziomych: 1 – ciąg pusty; 2 – 1, 2; 3 – 1, 2, 7, 10; 4 – 1, 2, 5, 10; 5 – 1, 10; 6 – 1, 8; 7 – 1, 3; 8 – 1, 12; 9 – 1, 4, 6, 12; 10 – 1, 4, 6, 9, 11, 12; 11 – 1, 4, 11, 12; 12 – 11, 12.

Możemy teraz przystąpić do opisania algorytm wyznaczania prostokątów z podziału wielokąta W. Prostokąty te wyznaczamy przeglądając kolejno boki pionowe w porządku zgodnym z $\propto$. Dla każdego boku b „ otwieramy" prostokąty, dla których b jest otwierający, i „zamykamy" prostokąty, dla których b jest zamykający. Prostokąty numerujemy zarówno w kolejności otwierania jak i zamykania. Numeracja ta zostanie wykorzystana do zdefiniowania grafu prostokątów. Ponieważ boki pionowe są przetwarzane zgodnie z porządkiem $\propto$, kolejność otwierania prostokątów jest zgodna z porządkiem $\propto$ dla ich lewych boków, a kolejność ich zamykania jest zgodna z porządkiem $\propto$ dla ich prawych boków.

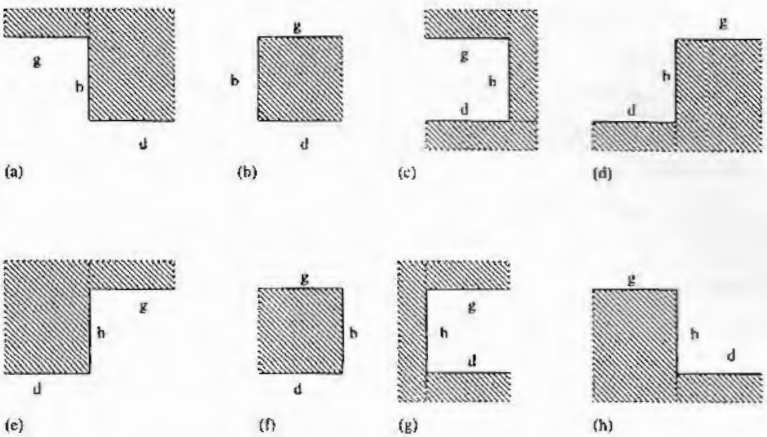
Otwarcie prostokąta polega na nadaniu mu numeru otwarcia i zapamiętaniu numeru linii pionowej siatki zawierającej lewy bok tego prostokąta (czyli współrzędnej x dolnego lewego rogu). Zamknięcie prostokąta polega na nadaniu mu numeru zamknięcia, policzeniu współrzędnych jego dolnego lewego rogu oraz jego wysokości i szerokości (czyli faktycznie wyznaczeniu prostokąta).

Pokażemy teraz w jaki sposób jest przetwarzany pionowy bok b wielokąta. Niech P będzie przekrojem dla b. W programie ZAM.PAS, przekrój ten jest oznaczony przez `Przekroj`. Przekrój P będzie aktualizowany w taki sposób, aby po przetworzeniu b, był on poprawny dla następnego po b boku w porządku $\propto$. Niech d i g będą sąsiednimi bokami b w wielokącie W, odpowiednio, o jednym z końców równym dolnemu końcowi b i o jednym z końców równym górnemu końcowi b.

Algorytm przetwarzania b zależy od statusu boków b, d, g: otwierający – zamykający. Mamy zatem do rozpatrzenia 8 przypadków.

(a) b-otw., d-otw., g-otw. (zob. rys. 5(a))
Znajdź w P bok $a = Nast(g)$.
Zamknij prostokąt ograniczony z dołu przez g i z góry przez a.
Jeśli prawy koniec a leży na linii siatki zawierającej b, to usuń a z P.
Usuń g z P.
Wstaw do P bok d.
Otwórz prostokąt ograniczony z dołu przez d.

(b) b-otw., d-otw., g-zam. (rys. 5(b))
Wstaw do P boki d i g.
Otwórz prostokąt ograniczony z dołu przez d.



Rysunek 5. Możliwe układy trzech kolejnych boków wielokąta.
Bok pionowy jest bokiem środkowym.

(c) b-otw., d-zam., g-otw. (rys. 5(c))
Jeśli b jest zamykający dla prostokąta ograniczonego z góry przez bok d, to
- znajdź w P bok $a = Poprz(d)$;
- zamknij prostokąt ograniczony z dołu przez a i z góry przez d;
- otwórz prostokąt ograniczony z dołu przez a, dla którego b jest otwierający;
- usuń d z P.

Znajdź w P bok $a = Nast(g)$.
Zamknij prostokąt ograniczony z dołu przez g i z góry przez a.
Jeśli prawy koniec a leży na linii siatki zawierającej b, to usuń a z P.
Usuń g z P.

(d) b-otw., d-zam., g-zam. (rys. 5(d))
Jeśli b jest zamykający dla prostokąta ograniczonego z góry przez bok d (wówczas jest otwierającym dla prostokąta ograniczonego z góry przez g), to
- znajdź w P bok $a = Poprz(d)$;
- zamknij prostokąt ograniczony z dołu przez a i z góry przez d;
- usuń d z P;
- otwórz prostokąt ograniczony z dołu przez a.

Wstaw bok g do P.

(e) b-zam., d-otw., g-otw. (rys. 5(e))
Znajdź w P bok $a = Nast(d)$.
Zamknij prostokąt ograniczony z dołu przez d i z góry przez a.
Jeśli prawy koniec a leży na linii siatki zawierającej b, to usuń a z P.
Usuń d z P.

Otwórz prostokąt ograniczony z dołu przez g, dla którego b jest otwierający.
Wstaw do P bok g

(f) b-zam., d-otw., g-zam. (rys. 5(f))
Zamknij prostokąt ograniczony z dołu przez d i z góry przez g.
Usuń z P boki d i g.

(g) b-zam., d-zam., g-otw. (rys. 5(g))
Jeśli b jest zamykający dla prostokąta sąsiadującego z b z lewej strony (jednocześnie jest wtedy otwierającym dla prostokąta ograniczonego z góry przez d), to
- znajdź bok $a = Poprz(d)$;
- znajdź bok $c = Nast(d)$;
- zamknij prostokąt ograniczony z dołu przez a i z góry przez c;
- otwórz prostokąt ograniczony z dołu przez a.
- jeśli prawy koniec c leży na linii siatki zawierającej b, to usuń c z P.

Otwórz prostokąt ograniczony z dołu przez g.
Wstaw d i g do P.

(h) b-zam., d-zam., g-zam. (rys. 5(h))
Jeśli b jest zamykający dla prostokąta ograniczonego z góry przez bok g (wówczas jest otwierającym dla prostokąta ograniczonego z góry przez d), to
- znajdź bok $a = Poprz(g)$;
- zamknij prostokąt ograniczony z dołu przez a i z góry przez a;
- usuń g z P;
- otwórz prostokąt ograniczony z dołu przez a.

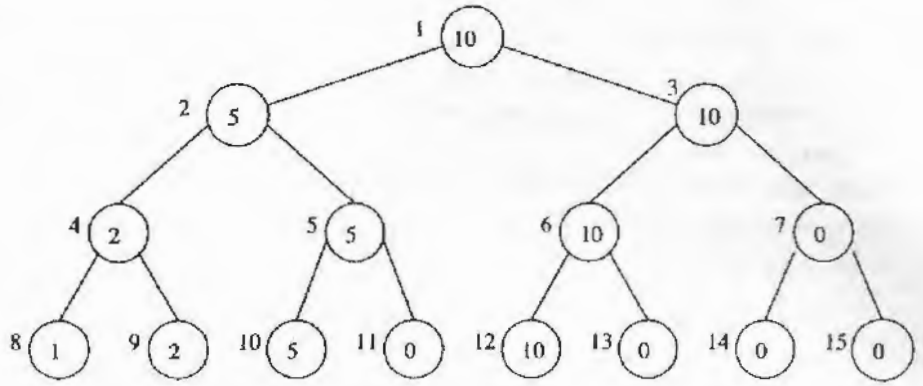
Wstaw bok d do P.

Opiszemy teraz operacje otwierania i zamykania prostokąta.

Zauważmy, że lewy bok otwieranego prostokąta jest zawsze zawarty w linii, do której należy bok b. Dodatkowo dany jest zawsze bok poziomy, który zawiera dolny bok otwieranego prostokąta. Niech e będzie takim bokiem (w zależności od przypadku $e = a$, d lub g).

Otwarcie prostokąta polega na zapamiętaniu (używamy do tego zmiennej *LewyBok*[e]) numeru linii zawierającej bok b, a tym samym numeru linii zawierającej lewy bok otwieranego prostokąta. Dodatkowo, zmiennej *Otw*[e] przypisujemy numer tego prostokąta w kolejności otwierania (te zmienne, oraz zmienne występujące w dalszym opisie rozwiązania mają w programie ZAM.PAS identyczne nazwy).

Zamykanie prostokąta polega na wyznaczeniu kolejnego prostokąta podziału. Wyznaczone prostokąty numerujemy w kolejności zamykania. Każdy prostokąt jest identyfikowany przez ten numer. Zauważmy, że prawy bok zamykanego



Rysunek 6. Implementacja przekroju dla boku 4 z wielokąta z rysunku 5.

prostokąta jest zawsze zawarty w linii zawierającej b. Dodatkowo, zawsze znany jest poziomy bok e ograniczający zamykany prostokąt z dołu. Zauważmy teraz, że współrzędne dolnego lewego rogu zamykanego prostokąta dane są przez *LewyBok*[e] (wsp. x) i numer linii poziomej zawierającej e (wsp. y). Szerokość prostokąta jest równa różnicy numeru linii zawierającej b i wartości *LewyBok*[e]. Aby znaleźć wysokość prostokąta wystarczy znać *Nast*(e). Wysokość jest różnicą numerów, linii zawierających *Nast*(e) i linii zawierającej e. Tablicę *Otw* wykorzystujemy do wyznaczenia porządku prostokątów w kolejności otwierania. Porządek ten pamiętamy w tablicy *PorzOtw*. Dokładniej, jeśli *NrZam* jest numerem zamykanego prostokąta, to w *PorzOtw*[*Otw*[e]] pamiętamy identyfikator zamykanego prostokąta *NrZam*.

Musimy teraz rozstrzygnąć w jaki sposób reprezentować przekrój P, a co za tym idzie w jaki sposób wykonywać na nim operacje: dodawania i usuwania boków oraz wyznaczania dla danego boku a, boków *Nast*(a) i *Poprz*(a). (W programie ZAM.PAS, *Nast* i *Poprz* są zrealizowane w postaci funkcji.) Przekrój można reprezentować na wiele sposobów, np. za pomocą listy. Podamy prostą implementację, która umożliwia bardzo efektywne wykonywanie wspomnianych operacji. Niech lw będzie liczbą warstw w wielokącie W. Zauważmy, że w każdym kroku obliczeń, z każdej warstwy do przekroju może należeć co najwyżej jeden bok. Przekrój będziemy pamiętali za pomocą pełnego drzewa binarnego ukrytego w tablicy *Przekrój*[1..2ll − 1], gdzie $ll = \min\{h : 2^h \geq lw, h = 0, 1, \ldots\}$ (zob. rys. 6).

Korzeniem drzewa jest *Przekrój*[1], a liśćmi węzły *Przekrój*[ll], *Przekrój*[ll + 1], ..., *Przekrój*[$ll + lw$ − 1]. Dla każdego i, $i = 1, \ldots, ll - 1$, synami węzła *Przekrój*[i]

są węzły *Przekrój*[2*i*] i *Przekrój*[2*i* + 1]. Dla każdego i, $i = 2,\ldots, 2ll - 1$, ojcem węzła *Przekrój*[*i*] jest węzeł *Przekrój*[$\lfloor i/2 \rfloor$]. Wysokość drzewa wynosi $\lceil \log lw \rceil$. Dla każdego $i = 1, \ldots, lw$, i-ty liść w drzewie (*Przekrój*[$ll + i - 1$]) reprezentuje i-tą warstwę wielokąta W. Na ostatnim poziomie drzewa (w liściach) warstwa położona niżej poprzedza warstwę położoną wyżej. Jeśli w aktualnym przekroju znajduje się bok poziomy z i-tej warstwy, to *Przekrój*[$ll + i - 1$] jest równe identyfikatorowi poziomego boku z i-tej warstwy aktualnie znajdującego się w przekroju, w przeciwnym razie wartością *Przekrój*[$ll + i - 1$] jest 0. Dla każdego $i = 1, \ldots, ll - 1$, wartością *Przekrój*[*i*] jest 0, jeśli do aktualnego przekroju nie należy żaden bok z warstw znajdujących się w liściach podrzewa o korzeniu w tym węźle. W przeciwnym razie wartością *Przekrój*[*i*] jest identyfikator boku z aktualnego przekroju leżący w najwyższej warstwie z poddrzewa o korzeniu w tym węźle. Taka reprezentacja umożliwia wykonywanie wspomnianych operacji na przekroju w czasie $O(\log lw)$. Nie opisujemy tutaj tych operacji dokładnie, a polecamy lekturę odpowiednich fragmentów programu ZAM.PAS.

Zwróćmy jeszcze uwagę, że powyższa reprezentacja przekroju umożliwia łatwe sprawdzanie, czy bok b jest bokiem zamykającym odpowiednich prostokątów (przypadki c, d, g, h). Rozważmy dla przykładu przypadek h (Pozostałe przypadki rozpatruje się analogicznie.). W tym przypadku wystarczy znaleźć bok $a = Poprz(g)$ i sprawdzić, czy wartość *LewyBok*[a] jest mniejsza od numeru linii siatki zawierającej b.

Pozostaje jeszcze problem wyznaczenia warstw wielokąta W. Można to łatwo zrobić ustawiając boki poziome w porządku $\propto$ zdefiniowanym wcześniej, a następnie licząc ile jest grup różnych elementów w tym porządku. Wiedząc ile jest warstw w wielokącie W można już w czasie $O(lw)$ zainicjalizować tablicę *Przekrój*[1..2ll − 1] (po prostu ją zerując).

Podsumowując, algorytm wyznaczania prostokątów podziału wielokąta W jest następujący:

1. Posortuj boki poziome W zgodnie z realcają $\propto$.
2. Dla każdego boku poziomego oblicz numer warstwy, do której ten bok należy i jednocześnie oblicz liczbę warstw lw. Oblicz ll.
3. Zainicjalizuj tablicę *Przekrój*[1..2ll − 1] wypełniając ją zerami.
4. Posortuj boki pionowe W zgodnie z $\propto$.
5. Dla kolejnych pionowych boków b w porządku $\propto$ zamknij prostokąty, które zamyka b i otwórz prostokąty, które otwiera b.

W programie ZAM.PAS, do sortowania boków używamy algorytmu HeapSort, który sortuje ciągi długości k w czasie $O(k \log k)$. Ponieważ liczba boków poziomych (pionowych) jest równa $n/2$, a liczba warstw nie może przekroczyć liczby boków poziomych, to z naszej dyskusji wynika, że wszystkie prostokąty

w podziale można wyznaczyć w czasie $O(\max(n \log n, m))$, gdzie m jest liczbą prostokątów w podziale. Jako ćwiczenie pozostawiamy do udowodnienia, że liczba prostokątów w podziale podłużnym nie przekracza $n/2 - 1$. Zatem, czas wyznaczenia wszystkich prostokątów w podziale wynosi $O(n \log n)$.

2. Zbuduj drzewo prostokątów T dla podziału z kroku 1.

Drzewo prostokątów budujemy w ten sposób, że dla każdego prostokąta A stworzymy listę prostokątów z nim sąsiadujących. Niech $P_1, P_2, \ldots, P_{lpr}$ będzie ciągiem prostokątów podziału zgodnym z kolejnością ich zamykania, a $Q_1, Q_2, \ldots, Q_{lpr}$ ciągiem tych samych prostokątów zgodnym z kolejnością ich otwierania. Można zauważyć, że $P_1, P_2, \ldots, P_{lpr}$ jest ciągiem prostokątów uporządkowanym zgodnie z relacją $\propto$ dla ich prawych boków, a $Q_1, Q_2, \ldots, Q_{lpr}$ jest ciągiem uporządkowanym zgodnie z relacją $\propto$ dla ich lewych boków. Ciągi te zostały wyznaczone w kroku 1. Wyznaczanie drzewa T jest podobne do scalania dwóch ciągów uporządkowanych. Korzystamy z faktu, że dla każdych dwóch prostokątów P_i, Q_j jeżeli prawy bok P_i i lewy bok Q_j mają wspólne punkty, to

- dla każdego $i' > i$, jeśli prawy bok $P_{i'}$ ma punkty wspólne z lewym bokiem $Q_{j'}$, to $j' \geq j$,
- dla każdego $j' > j$, jeśli lewy bok $Q_{j'}$ ma punkty wspólne z prawym bokiem $P_{i'}$, to $i' \geq i$.

Powyższy fakt gwarantuje, że drzewo prostokątów T można zbudować w czasie proporcjonalnym do liczby prostokątów podziału. Szczegółowa realizacja tego kroku jest zawarta w procedurze `BudujDrzewo`.

3. Wyznacz prostokąty *Pr(We)* i *Pr(Sk)* w podziale *Podz*.

Krok ten nie nastręcza żadnych trudności. Jest on realizowany za pomocą procedury, która w ciągu prostokątów znajduje pierwszy prostokąt zawierający podany punkt. Czas potrzebny na wykonanie tego kroku jest proporcjonalny do liczby prostokątów w podziale.

4. W drzewie T znajdź jedyną drogę $A_1 = Pr(We), A_2, \ldots, A_k = Pr(Sk)$, łączącą *Pr(We)* z *Pr(Sk)*.

Drogę $A_1, A_2, \ldots, A_k$ łączącą *Pr(We)* z *Pr(Sk)* znajdujemy przeszukując drzewo T w głąb (opis przeszukiwania w głąb można znaleźć w [4]). Koszt znalezienia drogi jest proporcjonalny do rozmiaru drzewa, czyli do liczby prostokątów w podziale.

5. Korzystając z drogi $A_1, A_2, \ldots, A_k$ w drzewie oblicz długość najkrótszej drogi łączącej *We* z *Sk* zgodnie ze wzorem (1).

Niech A i B będą sąsiednimi prostokątami podziału i niech punkt P będzie punktem zawartym w A. Znając współrzędne dolnych lewych wierzchołków prostokątów A i B oraz ich wysokości i szerokości można łatwo w czasie stałym wyznaczyć punkt Q leżący na wspólnym brzegu A i B, który leży najbliżej P. Stąd wynika, że długość najkrótszej drogi z *We* do *Sk* można obliczyć wyznaczając kolejno punkty $P_0 = We, P_1, P_2, \ldots, P_{k-1}, P_k = Sk$, gdzie P_i jest punktem położonym na wspólnym brzegu prostokątów A_i i A_{i+1} najmniej odległym od P_{i-1}, $i = 1, \ldots, k-1$, i sumując odległości $d_{A_i}(P_{i-1}, P_i)$, dla $i = 1, 2, \ldots, k-1$. Na końcu, do otrzymanej sumy należy dodać jeszcze $d_{A_k}(P_{k-1}, P_k)$. Powyższe obliczenia można wykonać w czasie proporcjonalnym do liczby prostokątów w podziale.

Pierwszy krok algorytmu jest wykonywany w czasie $O(n \log n)$, natomiast wszystkie pozostałe – w czasie liniowym $O(n)$. Stąd łączny koszt wykonania całego algorytmu wynosi $O(n \log n)$.

Omówienie rozwiązań podanych przez uczniów

To zadanie sprawiło zawodnikom wiele trudności. Tylko jeden z nich (Piotr Zieliński) uzyskał maksymalną liczbę punktów. Nie oznacza to, że pozostałe rozwiązania były błędne. W porównanin jednak z najlepszym rozwiązaniem były mniej efektywne. Idea algorytmu podanego przez P. Zielińskiego była podobna do przedstawionej przez nas. Użył on również pełnego drzewa binarnego do reprezentowania przekroju (chociaż nie posługiwał się określeniem „przekrój"). Złożoność jego rozwiązania była również $O(n \log n)$. Inni uczniowie stosowali również metodę podziału wielokąta na prostokąty. W większości jednak tych rozwiązań do reprezentowania „ przekroju" używano liniowych struktur danych takich, jak lista i tablica. Powodowało to, że złożoność tych rozwiązań była gorsza, rzędu $O(n^2)$.

Inne, często spotykane rozwiązanie, polegało na poruszaniu się od punktu wejścia do punktu ukrycia skarbu po brzegu wielokąta i dokonywaniu skrótów, gdzie tylko było to możliwe. Stosując tę metodę należy bardzo uważać, ponieważ nie zawsze wierzchołki optymalnej łamanej muszą leżeć na brzegu wielokąta. Ponadto, ze względu na skomplikowane kształty wielokątów, dokonywanie skrótów wymaga rozpatrywania wielu przypadków, w których można łatwo się pomylić.

Jeszcze inne rozwiązania polegały na redukcji zadania do zadania o rozmiarze mniejszym. W tym celu odcinano „ zaułki" zamku przesuwając, w razie konieczności, punkt startowy lub końcowy na brzeg zaułka. Wykrywanie zaułków okazało się bardzo kosztowne i zazwyczaj takie rozwiązania działały dość długo dla danych o dużych rozmiarach.

Jedno z najprostszych rozwiązań polegało na zbudowaniu grafu, którego wierzchołkami są punkty siatki należące do zamku i w którym krawędzie łączą sąsiednie punkty siatki. Po zbudowaniu takiego grafu, rozwiązanie zadania sprowadza się do znalezienia w nim najkrótszej drogi. Wadą tego rozwiązania jest to, że budowany graf może zawierać 100 000 000 wierzchołków, co czyni je mało praktycznym.

Testy

Do sprawdzania rozwiązań użyto 11 testów ZAM0.IN, ..., ZAM10.IN. Siedem pierwszych testów miało na celu sprawdzenie poprawności rozwiązań.

ZAM0.IN – test z treści zadania.

ZAM1.IN – trochę bardziej skomplikowany zamek, bardziej rozgałęziony i zawierający więcej wierzchołków.

ZAM2.IN – spory zamek z wieloma „zaułkami".

ZAM3.IN – zamek testujący algorytmy podziału na prostokąty i wymuszający złożone operacje na prostokątach.

ZAM4.IN – test podobny do poprzedniego, ale większy. Krawędzie są wyjątkowo złośliwie ułożone, po wiele o tej samej współrzędnej x i y, skomplikowane układy pionowe, poziome i mieszane

ZAM5.IN – prosty wielokąt, w którym długość optymalnej drogi jest bardzo duża (wymaga użycia typu `longint` w języku Turbo Pascal).

ZAM6.IN – test sprawdzający prawidłowe wykonywanie skrótów.

Ostatnie 4 testy służyły do sprawdzenia szybkości działania algorytmów. We wszystkich tych testach wielokąty miały co najmniej 4900 wierzchołków.

7.1.2. Zadanie PRĘTY (Autor: Piotr Chrząstowski-Wachtel)

Treść zadania PRĘTY

W laboratoriach pewnej firmy w czasie prac doświadczalnych nad nowym materiałem nazwanym politoksyparenem odkryto jego ciekawą właściwość. Otóż wykonany z tego materiału prosty pręt o małym przekroju, przy odpowiednim unieruchomieniu końców, po podgrzaniu wydłuża się i wygina dokładnie w łuk okręgu oparty na cięciwie, pokrywającej się z początkowym położeniem pręta. Załóżmy, że do doświadczeń potwierdzających tę właściwość użyto n prętów o zaniedbywalnie małym przekroju i początkowych długościach l_i ($1 \le l_i \le 100000$), oraz że w wyniku podgrzania wydłużyły się one odpowiednio o d_i ($1 \le d_i \le 100$), przy czym $d_i \le l_i/2$. Wszystkie wielkości są wyrażone w milimetrach.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego PRE.IN liczbę prętów n, a następnie ich długości l_i oraz wydłużenia d_i,
- dla każdego pręta oblicza odległość w mm, na jaką odchylił się po podgrzaniu środek pręta od swojego pierwotnego położenia (zakładamy, że pręt ma pomijalnie mały przekrój oraz że w końcowym położeniu przyjął kształt łuku opartego na cięciwie odpowiadającej początkowemu położeniu); wynik obliczenia ma być liczbą całkowitą różniącą się nie o więcej niż o 0.5 od dokładnej wartości odchylenia,
- zapisuje wynik do pliku tekstowego PRE.OUT.

Wejście

W pierwszym wierszu pliku tekstowego PRE.IN jest zapisana jedna liczba całkowita dodatnia $n \leq 50000$.

W każdym z kolejnych n wierszy są zapisane dwie liczby całkowite oddzielone pojedynczym odstępem – pierwotna długość kolejnego pręta l_i oraz jego wydłużenia d_i.

Wyjście

W każdym z n kolejnych wierszy pliku tekstowego PRE.OUT należy zapisać jedną liczbę całkowitą nieujemną – odchylenie odpowiedniego pręta obliczone z żądaną dokładnością.

Przykład

Dla pliku PRE.IN:

```
2
1000 20
15000 10
```

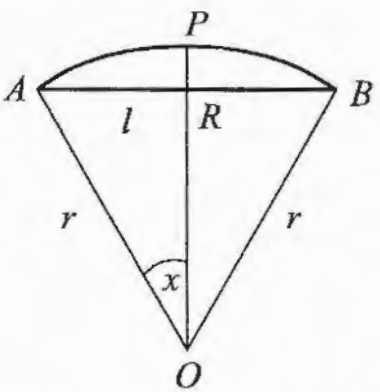
poprawnym rozwiązaniem jest plik PRE.OUT:

```
87
237
```

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

Rozwiązanie zadania PRĘTY

Przyjmijmy $l = l_i/2$, $d = d_i/2$ dla ustalonego i, $1 \le i \le n$. Niech O oznacza środek okręgu, którego łukiem jest pręt po odkształceniu, a A i B oznaczają końce łuku. Obierzmy punkt P na łuku AB tak, aby odcinek OP połowił kąt AOB. Oznaczmy przez R punkt przecięcia odcinków AB i OP, przez r promień okręgu, a przez x miarę łukową kąta AOP. Zobacz rys. 1.



Rysunek 1.

Mamy wtedy następujące zależności:

$$rx = l + d,$$
$$r \sin x = l.$$

Pierwsza zależność wynika z definicji miary łukowej, a druga – z trójkąta AOR.

Dzieląc równania stronami uzyskujemy

$$x/\sin x = (d + l)/l \tag{1}$$

czyli

$$x = \sin x\,(d + l)/l.$$

Należy rozwiązać równanie (1) ze względu na x. Znając x, szukaną wartość PR uzyskamy bez kłopotu, obliczając r np. z pierwszego równania, a następnie wstawiając r do równania $PR = OP - OR = r - r\cos x$. Oznaczmy przez $PR(x)$ wartość PR uzyskaną dla kąta x.

Zadanie sprowadza się więc do obliczenia x. Cóż prostszego? – wydawałoby się. Czeka nas jednak niespodzianka. Okazuje się, że równanie, które otrzymaliśmy, nie ma **rozwiązania elementarnego**, czyli wyrażającego się za pomocą funkcji będącej złożeniem skończonej liczby funkcji należących do jednej z następujących klas:

1. funkcje wymierne (ilorazy wielomianów),
2. funkcje potęgowe (w tym mieści się też pierwiastkowanie),
3. funkcje trygonometryczne,
4. funkcje wykładnicze,

oraz funkcje do nich odwrotne (czyli wszystkie arcusy i logarytmy).

Na dobrą sprawę, na lekcjach matematyki w szkole nie wychodzimy poza klasę funkcji elementarnych i rzeczywiście większość praktycznych zadań ma rozwiązanie wyrażające się za pomocą takich właśnie funkcji. Istnienie problemów, które nie mają elementarnych rozwiązań często umyka naszej uwadze. Przedstawienie takiego problemu było celem ubocznym przy wyborze tego zadania.

Powstają dwa pytania: jak rozpoznać, czy dla danego zadania (równania) istnieje rozwiązanie elementarne, oraz co należy zrobić, jeśli nie istnieje? Odpowiedź na pierwsze pytanie nie jest łatwa. Ogólnie rzecz biorąc, gdy z jednej strony równania znajduje się funkcja innego typu (spośród 1–4) niż z drugiej, wówczas na ogół nie da się elementarnie rozwiązać takiego równania, chyba że jest ono jakiejś szczególnej postaci. Na przykład, choć równanie $\sin x = x$ ma przez przypadek proste rozwiązanie elementarne ($x = 0$), to równania $\cos x = x$ nie da się rozwiązać w elementarny sposób, choć rozwiązanie niewątpliwie istnieje, o czym się każdy może przekonać rysując wykresy funkcji $y = x$ i $y = \cos x$. Wiadomo również (udowodnił to wielki matematyk norweski Niels Abel na początku XIX wieku), że równania w klasie wielomianów wyższego stopnia niż 3 zazwyczaj nie mają rozwiązań elementarnych, choć i to zależy od postaci wielomianu. Tym, którzy w to nie wierzą proponujemy rozwiązać równanie $x^5 - x = 1$.

Odpowiedź na drugie pytanie jest nieco prostsza. Jeżeli nie da się rozwiązać zadania dokładnie, przedstawiając jego rozwiązanie w postaci funkcji elementarnej, to należy przybliżyć je możliwie dobrze. W końcu, nawet jeśli otrzymalibyśmy odpowiedź typu $\log_2 5$, to i tak niewiele moglibyśmy z tym zrobić poza znalezieniem odpowiednio dobrego przybliżenia pozycyjnego (dziesiętnego lub dwójkowego). Gdy liczymy na komputerze i używamy skończonych dziesiętnych ułamków takich jak 0.22 czy nawet 0.1, wówczas i tak reprezentacja binarna tych liczb jest niedokładna: w układzie dwójkowym mają one nieskończone okresowe rozwinięcia, które w pewnym miejscu się po prostu ucina, zaokrąglając resztę do ostatniego bitu.

Przybliżanie, czyli **aproksymacja** rozwiązania wydaje się być zatem całkiem naturalnym rozwiązaniem naszego problemu. Szczególnie, że rozwiązań poszukujemy z zaokrągleniem do najbliższej liczby całkowitej; nie interesują nas zatem zbyt dalekie cyfry po kropce dokładnego rozwiązania. Tu pojawia się pewien kłopot, którego żaden z uczestników nie zauważył. Gdyby któreś z roz-

wiązań było bardzo bliskie połowy między dwiema kolejnymi liczbami naturalnymi, wówczas powstałby problem, w którą stronę je zaokrąglić: w górę, czy w dół. W końcu, w wyniku naszych obliczeń, moglibyśmy otrzymać coś w rodzaju 13.4999998, ale błąd zakumulowany w trakcie obliczeń mógłby wypaczyć obraz dokładnego rozwiązania, które mogłoby być np. położone bliżej 13.5000002. W wyniku naszych obliczeń zaokrąglilibyśmy wynik w dół otrzymując w odpowiedzi 13, podczas gdy prawidłowo powinniśmy byli zaokrąglić wynik w górę uzyskując w odpowiedzi liczbę 14.

Na szczęście, w podanym zakresie danych nie było przypadku, żeby wynik leżał na tyle blisko połowy między dwiema liczbami naturalnymi, aby mogły wyniknąć stąd jakieś kłopoty numeryczne. Gdyby się okazało, że dla jakiejś pary l i d wynik wypada bliżej połowy między dwiema liczbami całkowitymi niż np. 10^{-13}, wówczas określenie z której strony połowy znajduje się rozwiązanie byłoby trudne i wymagałoby przynajmniej użycia podwójnej precyzji. W naszym przypadku obliczenia wykonywane nawet w standardowej precyzji i korzystanie z wbudowanych funkcji trygonometrycznych wystarczały do poprawnego rozwiązania zadania.

Metod rozwiązywania tego typu równań (zwanych **nieliniowymi**) jest wiele. Moglibyśmy zastosować w tym przypadku np. metodę Newtona, opisaną w podręczniku [11] (zob. również demonstracje i realizacje różnych metod rozwiązywania równań nieliniowych w pakiecie EI [10]). Musielibyśmy wtedy przeprowadzić analizę zbieżności tej metody, gdyż metoda ta nie zawsze jest skuteczna i czasami oddalamy się od rozwiązania, zamiast się do niego przybliżać.

Prostszą koncepcyjnie metodą jest – stosowana zresztą przez większość zawodników – **metoda bisekcji**, która polega na stopniowym zawężaniu przedziału, w którym możemy się spodziewać rozwiązania. W tym celu ustalamy najpierw dwie wartości startowe, a oraz b, o których możemy powiedzieć, że pierwsza z nich jest na pewno mniejsza, niż rozwiązanie, a druga na pewno większa, czyli $[a, b]$ jest przedziałem, w którym leży poszukiwane rozwiązanie. Następnie wybieramy środek przedziału $s = (a + b)/2$ i badamy, czy s jest większe od rozwiązania, czy mniejsze, i w zależności od odpowiedzi wykonujemy przypisanie albo $b := s$, albo $a := s$, utrzymując stale spełniony niezmiennik, że rozwiązanie znajduje się pomiędzy a i b. Postępowanie takie można stosować w przypadku funkcji ciągłych, a z takimi mamy tu niewątpliwie do czynienia.

Pozostaje problem, kiedy zakończyć algorytm bisekcji. Zazwyczaj, połowienie przedziału kontynuujemy tak długo, aż uzyskamy wymaganą precyzję, albo aż kolejne obroty pętli niczego nie zmieniają. W naszym przypadku będziemy mo-

gli przerwać obliczenia nieco szybciej. O tym za chwilę, a na razie zastanówmy się, jak metodę bisekcji zastosować w przypadku naszego zadania.

Zauważmy, że zadanie odwrotne rozwiązuje się bardzo prosto. Mając dany kąt x, na którym jest oparty łuk AP i długość l możemy znaleźć wydłużenie d korzystając z przekształconego wzoru (1):

$$d = lx/\sin x - l. \tag{2}$$

Mamy więc już wszystko, czego nam potrzeba. Obiektem naszego zaiuteresowania będzie kąt x i ten kąt właśnie będziemy przybliżali za pomocą bisekcji. Za wartości początkowe końców przedziału możemy np. przyjąć $a = 0$ oraz $b = \pi\, l$ – na pewno dla takich kątów otrzymane wartości d ze wzoru (2) będą odpowiednio mniejsze i większe od wartości d, którą znamy z treści zadania. Dla kolejno badanych $s = (a + b)/2$ będziemy określali, czy prawa strona równania (2) dla badanego $x = s$ przewyższa d, czy jest od niego mniejsza, i w zależności od tego będziemy przesuwali albo a albo b. Kiedy powinniśmy skończyć tę pętlę?

Ponieważ długość przedziału, w którym poszukujemy wartości x, czyli $b - a$, w każdym kroku pętli zmniejsza się o połowę, więc algorytm nasz powinien się zatrzymać co najwyżej po tylu obrotach pętli, ile razy można przez 2 podzielić długość badanego przedziału zanim nie uzyskamy zera. Możemy jednak przerwać algorytm wcześniej, gdy tylko okaże się, że wartości $PR(a)$ i $PR(b)$ zaokrąglone do najbliższej liczby całkowitej dadzą ten sam wynik. W tym przypadku dalsze iteracje niczego już nie zmienią.

Warto jeszcze skomentować wyniki, które dostaniemy. Przeczą one czasami intuicji. Niektórzy uczniowie rozwiązawszy poprawnie zadanie szukali błędu, gdyż wydawało się im, że niemożliwe są aż takie wychylenia przy tak małych przyrostach długości. Podajmy dla przykładu kilka poprawnych odpowiedzi (w milimetrach) dla par (l_i, d_i).

l_i \ d_i	1	5	10	15	20
10	2	5	6	8	10
100	6	14	20	24	28
1000	19	43	61	75	87
10000	61	137	194	237	274
100000	194	433	612	750	866

Wydłużenie na przykład stumetrowego pręta o jeden centymetr powoduje więc jego wygięcie o mniej więcej 61.2 cm. Fenomen znacznego odchylenia przy niewielkich wydłużeniach względnych (tu wydłużenie względne wynosi 0.0001, a odchylenie względne wynosi około 0.00612, jest więc ponadsześćdziesięciokrot-

nie większe od wydłużenia względnego) jest przyczyną niebezpieczeństw związanych z wyginaniem się szyn w czasie upałów, kiedy niewielkie przyrosty długości szyny powodują bardzo widowiskowe odkształcenia zagrażające bezpieczeństwu jazdy pociągiem. Stąd tak duże zapasy projektowane w przerwach między szynami, które mają zniwelować ten efekt.

Proponujemy jeszcze sprawdzić, czy jeśli taśmę okalającą szczelnie Ziemię na równiku wydłużymy o 1 metr i odsuniemy od Ziemi na jednakową odległość, to przeciśnie się pod nią mysz?

Omówienie rozwiązań podanych przez uczniów

Poprawne rozwiązania tego zadania można podzielić na trzy grupy.

1. Rozwiązanie nieliniowego równania za pomocą bisekcji, podobnie jak to zostało opisane powyżej.
2. Rozwiązanie nieliniowego równania za pomocą metody Newtona. Metoda ta jest zazwyczaj szybsza od podanej metody bisekcji. Jeden z uczniów przyspieszył znacznie stosowanie tej metody tablicując wartości początkowe, od których warto zaczynać iteracje.
3. Użycie wzoru przybliżonego na poszukiwane wychylenie pręta. Wzór taki, zaczerpnięty np. z [8] ma postać:

$$h_p = \frac{\sqrt{3\,(l_i + d_i)^2 - 3l_i^2}}{4}$$

i definiuje przybliżoną wartość wychylenia. Wzór ten podaje dość dobre przybliżenie, znajdując odpowiednią wartość z dokładnością nie przekraczającą 1.5 dla danych z zadania. Taka dokładność nie wystarcza, aby zadanie poprawnie rozwiązać, niemniej skorygowanie wyniku o 1 pozwala namierzyć właściwą wartość całkowitą. Wystarczy bowiem sprawdzić, czy dla liczb całkowitych w otoczeniu h_p uzyskana wartość spełnia dokładne równanie

$$\sin\frac{4\,(l_i + d_i)\,h_p}{4\,h_p^2 + l_i^2} = \frac{4\,l_i\,h_p}{4\,h_p^2 + l_i^2},$$

a dokładniej, czy funkcja

$$f(h_p) = \sin\frac{4\,(l_i + d_i)\,h_p}{4\,h_p^2 + l_i^2} - \frac{4\,l_i\,h_p}{4\,h_p^2 + l_i^2}$$

przyjmuje wartość dodatnią, czy ujemną. Jeśli wartość ta jest dodatnia, to h_p jest zaniżone, jeśli jest ujemna, to h_p jest zawyżone. Powtarzając dodawanie bądź odejmowanie jedynki można znaleźć takie h_p, żeby wartość

$f(h_p + 1)$ miała przeciwny znak, niż wartość $f(h_p)$. Przyjęcie wartości h_p, dla której $f(h_p)$ da wynik bliższy zera, stanowi rozwiązanie zadania. Ten sposób działania dawał bardzo szybkie rozwiązanie.

Testy

Testy dla tego zadania były trzech typów: poprawnościowe, dokładnościowe i wydajnościowe. Testy poprawnościowe sprawdzały, czy algorytm poprawnie liczy proste przykłady. Testy dokładnościowe sprawdzały przede wszystkim dokładność obliczeń, a wydajnościowe, czy w przypadku dużego zestawu danych rozwiązanie nie zużywa zbyt dużo czasu.

Jak zwykle, pierwszy test PRE0.IN był testem z tekstu zadania. Kolejny test sprawdzał, czy skrajne wartości z dopuszczalnego przedziału zostały uwzględnione w rozwiązaniu. Dla danych:

```
2 1
100000 100
100000 1
```

rozwiązaniami są: 1, 1937, 194.

Następne trzy testy stanowiły losowe, takie „normalne" dane:

```
80000 2
200 12
80 30
```

o odpowiedziach równych odpowiednio: 245, 30, 32.

Test PRE7.IN stanowiła grupa trzech danych, dla których trzeba było osiągnąć dokładność do pierwszej cyfry po kropce, aby uzyskać poprawne zaokrąglenie. Były to dane:

```
20006 15
20133 3
42353 61
```

o odpowiedziach równych odpowiednio: 335, 151, 985.

Następny test grupował cztery zestawy wymagające precyzji do czwartego miejsca po kropce. Były to liczby

```
1212 10
5859 36
7901 46
8949 92
```

o odpowiedziach równych odpowiednio: 67, 282, 369, 557.

Ostatnia grupa testów dokładnościowych wymagała precyzji do 6. lub 7. miejsca po kropce. Były to liczby

27732 10
41414 47
56511 84
80565 94

o odpowiedziach równych odpowiednio: 322, 855, 1334, 1686. Ten ostatni test dawał w odpowiedzi, przy precyzji obliczeń rzędu 10^{-13}, liczbę 1685.50000009..., którą należało oczywiście zaokrąglić w górę.

Testy kończył zestaw 10000 w miarę losowych danych, który sprawdzał wydajność czasową algorytmu.

7.1.3. Zadanie MOKRA ROBOTA (Autor: Piotr Chrząstowski-Wachtel)

Treść zadania MOKRA ROBOTA

Dysponujemy n naczyniami, gdzie $1 \le n \le 4$. Wszystkie naczynia są początkowo całkowicie wypełnione wodą. Pojemność i-tego naczynia o_i, mierzona w litrach, jest liczbą naturalną spełniającą nierówności $1 \le o_i \le 49$.

Wolno wykonywać trzy rodzaje ruchów:

1. Przelanie całej zawartości z jednego naczynia do drugiego. Ruch ten można wykonać tylko wtedy, gdy w drugim naczyniu jest wystarczająco dużo wolnego miejsca.
2. Dolanie wody do pełna z jednego naczynia do drugiego.
3. Wylanie całej zawartości jednego naczynia do zlewu.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego MOK.IN liczbę naczyń n, pojemność każdego naczynia i zadaną końcową ilość wody w każdym z naczyń,
- bada, czy można w skończonej liczbie ruchów doprowadzić do zadanej sytuacji końcowej i jeśli tak, znajduje minimalną liczbę ruchów prowadzących do niej,
- zapisuje wynik, tj. słowo NIE lub minimalną liczbę ruchów w pliku tekstowym MOK.OUT.

Wejście

W pierwszym wierszu pliku tekstowego MOK.IN jest zapisana jedna liczba całkowita dodatnia $n \le 4$ – jest to liczba naczyń.

W drugim wierszu jest zapisanych n liczb naturalnych. Kolejna i-ta liczba $1 \le o_i \le 49$ jest pojemnością i-tego naczynia.

W trzecim wierszu jest zapisanych n liczb naturalnych. Kolejna i-ta liczba $0 \le w_i \le o_i$ jest zadaną końcową ilością wody w odpowiednim i-tym naczyniu.

Liczby w wierszach drugim i trzecim są pooddzielane pojedynczym odstępem.

Wyjście

Jeśli nie można doprowadzić do zadanej sytuacji końcowej, to w pierwszym i jedynym wierszu pliku tekstowego MOK.OUT należy zapisać jedno słowo NIE, a w przeciwnym przypadku minimalną liczbę ruchów prowadzących do zadanej sytuacji końcowej.

Przykłady

Dla pliku MOK.IN:

```
3
3 5 5
0 0 4
```

poprawnym rozwiązaniem jest następujący plik MOK.OUT:

```
6
```

Dla pliku MOK.IN:

```
2
20 25
10 16
```

poprawnym rozwiązaniem jest plik MOK.OUT:

```
NIE
```

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

Rozwiązanie zadania MOKRA ROBOTA

Rozwiązanie zadania powinno się zacząć od sprawdzenia pewnych warunków, które mogą wykluczyć istnienie rozwiązania. Zadanie jest dość złożone, a niektóre warunki sprawdza się błyskawicznie, więc ewentualne opóźnienie stąd wynikające w przypadku, gdy rozwiązanie istnieje, jest znakomicie kompensowane w przypadkach, gdy rozwiązanie nie istnieje z jakiegoś powodu, a my to błyskawicznie wykryjemy. W zadaniach tego typu bowiem sytuacje,

w których rozwiązania nie ma, są najbardziej złośliwe – zmuszają do przejrzenia pełnej przestrzeni możliwych stanów.

Jeden warunek jest oczywisty: rozwiązania na pewno nie ma, gdy liczba $d = \mathrm{NWD}(o_1, \ldots, o_n)$ nie dzieli któregoś z w_i. W takim przypadku bowiem początkowe ilości wody są podzielne przez d, a każda operacja przelania wody powoduje, że zarówno w naczyniu, z którego wylewamy wodę, jak i w naczyniu do którego ją wlewamy pozostaną ilości podzielne przez d. Sprawdzenie tego warunku zajmuje niewiele czasu – realizuje to funkcja `JestPodzielne` z modułu WSTEPNE.PAS.

Z warunków zadania wynika również, że każdy z ruchów musi pozostawić po sobie jedno naczynie puste lub jedno naczynie pełne. Jeśli więc konfiguracja końcowa nie spełnia tego warunku, to niewątpliwie osiągnąć jej nie można (por. funkcję `JestPusteLubPelne` z modułu WSTEPNE.PAS).

W przypadku, gdy nie wykluczymy istnienia rozwiązania, zadanie sprowadza się do problemu znalezienia najkrótszej drogi w grafie skierowanym, określonym w następujący sposób. Wierzchołkami grafu są wektory $(v_1, \ldots, v_n)$ dla $v_i < o_i$. Łuki w tym grafie łączą dwa jego wierzchołki $(a_1, \ldots, a_n)$ oraz $(b_1, \ldots, b_n)$ wtedy i tylko wtedy, gdy istnieje pojedynczy ruch przeprowadzający stan napełnienia naczyń $a_1, \ldots, a_n$ w stan $b_1, \ldots, b_n$. Zauważmy, że tak zdefiniowany graf może być bardzo duży. Potencjalnie, możliwych wierzchołków jest $o_1 \times \ldots \times o_n$, co przy warunkach ograniczających rozmiar zadania każe zachować dużą ostrożność. Gdybyśmy chcieli najpierw zbudować graf, a potem szukać w nim najkrótszej drogi którymś ze znanych algorytmów, skazani bylibyśmy najprawdopodobniej na niepowodzenie: całkiem prawdopodobne jest, że zabrakłoby pamięci w trakcie jego działania.

Graf należy tworzyć równolegle z szukaniem najkrótszej drogi prowadzącej do wierzchołka $(w_1, \ldots, w_n)$. Najstosowniejszą metodą wydaje się przechodzenie grafu wszerz, opisaną np. w rozwiązaniu zadania o Klubie Prawoskrętnych Kierowców ([2], str. 79–80). Początkowo kładziemy do kolejki wierzchołek $(o_1, \ldots, o_n)$ (por. procedurę `Inicjuj` z modułu KOLEJKA.PAS) i badamy wszystkie możliwe pierwsze ruchy, czyli w tym przypadku wylanie całej zawartości do zlewu kolejno z wszystkich naczyń (por. procedurę `Wylej` z modułu TYPY.PAS wywoływaną w funkcji `Osiagalna` z modułu KOLEJKA.PAS). Otrzymamy w ten sposób wierzchołki $(0, o_2, \ldots, o_n)$, $(o_1, 0, o_3, \ldots, o_n)$, …, $(o_1, \ldots, o_{n-1}, 0)$. Wkładamy je do kolejki z zaznaczeniem numeru ruchu dotyczącego wkładanego wierzchołka (w tym przypadku numer ten równa się 1 – tyle razy do tej pory przelewaliśmy cokolwiek). Następnie pobieramy z kolejki kolejno wierzchołki i dla każdego z nich, po sprawdzeniu czy nie jest to wierzchołek końcowy, wstawiamy do kolejki jego następniki (por. procedurę `Wstaw`

z modułu KOLEJKA.PAS) uważając, żeby jakiegoś wierzchołka nie powtórzyć. Sprawdzenie tego warunku, ze względu na złożoność struktury grafu, może nie być banalne. W tym celu można zastosować rozwiązanie klasycznego problemu słownika. **Problem słownika** polega na doborze takiej struktury danych, aby można było efektywnie wykonać następujące operacje na zbiorze:

1. Zainicjowanie pustego zbioru.
2. Dodanie do badanego zbioru elementu x.
3. Sprawdzenie, czy w badanym zbiorze znajduje się element x.
4. Usunięcie elementu x z badanego zbioru.

Zbiór z takimi operacjami nazywamy **słownikiem**.

Operacje te występują w rozwiązaniach wielu problemów. Znalezienie efektywnej implementacji słownika polega na takiej jego realizacji, aby każdą z powyższych operacji wykonać w czasie logarytmicznym ze względu na liczbę elementów znajdujących się aktualnie w słowniku.

Aby uporządkować przechowywaną część grafu warto rozdzielić te konfiguracje (wierzchołki), które mają różne łączne ilości wody. Taką łączną ilość wody określoną przez wierzchołek nazwiemy jego **wagą**. Zauważmy, że cały proces przelewania odbywa się w taki sposób, że przez jakiś czas (być może zerowy) przelewamy wodę między naczyniami, a potem wylewamy z jakiegoś naczynia całą wodę do zlewu, potem znowu przelewamy między naczyniami, znowu wylewamy do zlewu itd. Wierzchołki, które mają łącznie tyle samo wody, stanowią pewną klasę abstrakcji. Warto więc nie mieszać różnych wag wierzchołków; zaoszczędzi nam to trochę czasu kosztem niewielkiej dodatkowej pamięci: raptem będziemy potrzebowali 200 wskaźników do słowników reprezentujących różne wagi. Zakładamy więc istnienie tablicy słowników (tablica `Byly` w rekordzie `Kolej`), każdy z nich będzie odpowiedzialny za inną wagę. Poszukiwanie danego elementu będzie się więc składało z wyliczenia jego wagi w, a następnie przejrzenia w-tego słownika w celu otrzymania odpowiedzi na pytanie, czy ten element już rozważaliśmy. Jeżeli tak, to przerywamy działania dotyczące tego elementu, jeśli nie, to wstawiamy go do w-tego słownika (por. procednrę `WstawDoDrzewa` z modułu KOLEJKA.PAS) i, niezależnie, do kolejki (por. procedurę `WstawDoKolejki` z tego samego modułu). Taki zabieg wstępnego posegregowania elementów zgodnie z jakąś łatwo obliczalną własnością (tu – wagą) nazywamy **haszowaniem**. Schemat haszowania ma następującą postać:

1. Oblicz wartość funkcji haszującej, czyli funkcji, która dla każdego elementu daje pewną wartość indeksu w tablicy.
2. Pod otrzymanym indeksem sprawdź, czy nie ma tam poszukiwanego klucza. Jeżeli się zdarzy, że więcej niż jeden klucz trafia za pomocą funkcji haszującej na ten sam indeks, to zastosuj tzw. rehaszowanie, czyli procedurę

umożliwiającą lokalizację różnych elementów znajdujących się pod tym samym kluczem.

Rehaszowanie realizuje się zgodnie z przyjętą strukturą danych do zapamiętywania kluczy o tej samej wartości funkcji haszującej. Jeżeli jest to lista, to polega ono na przejrzeniu listy. Jeżeli jest to drzewo, to szukamy w drzewie itd. W naszym rozwiązaniu rehaszowanie sprowadza się do przejrzenia drzewa binarnych wyszukiwań, o czym później.

Manewr haszowania ma w naszym przypadku dodatkową, niebagatelną zaletę. Umożliwia bowiem zwolnienie pamięci zajmowanej przez martwe wierzchołki grafu – czyli takie, których następniki zostały już dodane i które nie mają szans już wystąpić w dalszych fazach algorytmu. Zauważmy bowiem, że nie można wyrzucać ze słownika elementu od razu po jego przetworzeniu: z dużym prawdopodobieństwem pojawiłby się z powrotem jako następnik innego wierzchołka, stwarzając możliwość zapętlenia się algorytmu. Wyobraźmy sobie bowiem uproszczoną sytuację, w której przelewamy jednostkę wody pomiędzy dwoma naczyniami. Wierzchołek (1,0) pobierany jest z kolejki (która na moment staje się pusta). Wierzchołek ten wstawia do kolejki i do słownika element (0,1) i sam usuwa się ze słownika. Element (0,1) działa analogicznie: usuwa się z kolejki i wstawia do kolejki i słownika element (1,0) itd.

Tak więc z usuwaniem ze słownika musimy być ostrożni. Tym niemniej możemy usunąć cały słownik o wadze w wtedy, gdy w kolejce znajdować się będą wyłącznie elementy o wagach mniejszych od w. Wody bowiem nie może łącznie przybyć. Taka sytuacja będzie oznaczała, że nie ma szans abyśmy kiedykolwiek poszukiwali elementów o wadze w i cały słownik stanie się wtedy niepotrzebny. Oczywiście znów trzeba być ostrożnym przy sprawdzaniu tego warunku. Przeglądanie całej kolejki byłoby bardzo kosztowne: może ona zawierać zbyt dużo elementów. Można jednak uprościć to zadanie wyposażając każdy słownik w licznik zainicjalizowany na zero. Licznik ten będziemy zwiększali o 1, ilekroć będziemy wstawiali nowy element do słownika, a zmniejszali o 1 – ilekroć pobierzemy z kolejki jakiś element z tego słownika. Gdy licznik największej niepustej wagi zostanie wyzerowany, to będzie to oznaczało, że więcej do tego słownika nikt nie zajrzy, więc cały słownik dotyczący tej wagi można będzie usunąć z pamięci.

W przedstawionym rozwiązaniu, pojedynczy słownik odpowiadający zadanej wadze zrealizujemy za pomocą drzewa binarnych wyszukiwań (**drzewa BST**). W tym celu wprowadzimy porządek leksykograficzny na konfiguracjach zawartości wody w naczyniach. Potraktujemy więc kolejne liczby w naczyniach, jako cyfry w układzie o podstawie `MaxPoj+1`. W naszym przypadku na przykład, dla 4 naczyń, gdzie maksymalna pojemność naczynia wynosi 49, konfiguracja

8 5 3 0 będzie kodowana za pomocą liczby $8 \times 50^3 + 5 \times 50^2 + 3 \times 50 + + 0 \times 50^0$. Ten sposób kodowania zapewnia jednoznaczność przyporządkowania liczby konfiguracji i umożliwia porównywanie konfiguracji ze sobą. W drzewie binarnych wyszukiwań panuje zasada, że w każdym węźle (**Uwaga.** Dalej, zgodnie z powszechnie przyjmowaną zasadą, **wierzchołek** oznacza element grafu, a **węzeł** oznacza element drzewa wyszukiwań w tym grafie.) znajduje się wartość większa od wszystkich wartości z lewego poddrzewa tego węzła, a mniejsza od wszystkich wartości z prawego poddrzewa.

Wstawianie do drzewa binarnych wyszukiwań jest bardzo proste: jeżeli drzewo jest puste, to trzeba utworzyć węzeł i wstawić żądaną wartość. Jeżeli drzewo jest niepuste, to trzeba porównać wstawianą wartość z tą, która znajduje się w korzeniu. Jeśli te wartości są sobie równe, to nie wstawiamy niczego, tylko sygnalizujemy tę sytuację (u nas to będzie oznaczało wystąpienie już badanego wierzchołka w słowniku). Jeśli wstawiana wartość jest mniejsza od tej w korzeniu, to rekurencyjnie wstawimy ją do lewego poddrzewa, jeśli jest większa, to do prawego.

Wykonanie słownikowej operacji usuwania pojedynczego elementu z drzewa binarnych wyszukiwań jest nieco bardziej skomplikowane, ale my pojedynczych elementów usuwać nie będziemy, więc zainteresowanych odsyłamy do książki [6]. Usuwanie całego drzewa, czyli to, czego tu potrzebujemy, wykonuje się w standardowy sposób (por. procedurę `UsunZbedne` z modułu KOLEJKA.PAS), jak dla każdego drzewa binarnego.

Pozostaje jeszcze problem wysokości tak tworzonego drzewa. W najbardziej niekorzystnym przypadku, gdy wstawiane elementy będą tworzyły ciąg monotoniczny, wysokość drzewa będzie równa liczbie wstawianych elementów. Przed takim przypadkiem można się zabezpieczyć **równoważąc** drzewo, na przykład algorytmem AVL, opisanym w [6]. W naszym przypadku kolejność dodawanych do słowników wierzchołków jest mniej więcej przypadkowa i w rezultacie tworzone drzewo nigdy zbyt wysokie nie jest. W przedstawionym rozwiązaniu nie dbamy więc o wysokość drzewa, ufając że nie wpłynie to w znaczącym stopniu na koszt algorytmu. Niewykluczone, że akurat dla tego zadania równoważenie drzewa, które też kosztuje, pogarsza wręcz średni koszt działania algorytmu. Tym niemniej warto pamiętać, że drzewa binarnych wyszukiwań można przetwarzać tak, aby nawet w najgorszym przypadku móc zrealizować na nich słownik o koszcie wykonania każdej z operacji słownikowych rzędu logarytm z liczby węzłów drzewa.

Podstawową strukturą danych w naszym algorytmie jest rekord `Element` (moduł KOLEJKA.PAS), reprezentujący wierzchołek grafu. Rekord ten zawiera dodatkowe informacje umożliwiające efektywne przetwarzanie omawianych

struktur. Poza samą konfiguracją `st:Stan`, określającą ilość wody w naczyniach, mamy pole `kr:longint` określające liczbę kroków, które nas przywiodły do tego wierzchołka, a także pola wskaźnikowe `Lewy`, `Prawy` definiujące wskaźniki do poddrzew w odpowiednim słowniku, jak również pole `Nast`, określające następny element w kolejce.

Sama kolejka jest rekordową strukturą danych, w której przechowujemy następujące informacje:

```
Pierw,Ost:PElement; {wskaźniki do pierwszego i ostatniego
                     elementu kolejki}
MaxWag    :-1..MaxWaga; {maksymalna waga elementu w kolejce}
Byly      :array[Waga] of
            record
             Ile :longint; {liczba elementów w kolejce o tej
                            samej wadze}
             Korz:PElement {korzenie słownikow o tej wadze}
            end
```

Pobierając zatem pierwszy element v_0 z kolejki, dla wszystkich następników v wierzchołka v_0 w rozważanym grafie wykonujemy kolejno następujące kroki:

1. Sprawdzamy, czy wierzchołek v nie jest wierzchołkiem docelowym i jeśli tak jest, to kończymy podając jako odpowiedź wartość v_0.kr+1. Jeśli tak nie jest, to wykonujemy kroki 2 – 4.
2. Obliczamy wagę w wierzchołka v.
3. Sprawdzamy, czy warto dodawać element v do kolejki. W tym celu badamy 2 warunki:
   a. czy waga wierzchołka v nie jest przypadkiem mniejsza, niż waga konfiguracji końcowej?
   b. czy w słowniku `Byly[`w`].korz` znajduje się już konfiguracja odpowiadająca wierzchołkowi v?
4. Jeśli odpowiedź na któreś z powyższych pytań jest pozytywna, to przetwarzanie wierzchołka v przerywamy i przechodzimy do kolejnego następnika, w przeciwnym razie:
   a. Wstawiamy konfigurację wierzchołka v do tego słownika.
   b. Wkładamy v na koniec kolejki, czyli wierzchołek v doczepiamy jako pole `Nast` elementu wskazywanego przez pole `Ost` kolejki, a następnie zmieniamy wartość pola `Ost` tak, aby wskazywało ono na v.
   c. Zwiększamy licznik `Byly[`w`].Ile` o 1.

Na zakończenie przetwarzania wierzchołka v_0 zmniejszamy wartość licznika `Byly[Waga(`v_0`)].Ile` o 1. Sprawdzamy teraz, czy przypadkiem nie wyzerowaliśmy tego licznika i czy jednocześnie `Waga(`v_0`)` nie była największą rozważaną w tym momencie wagą. Jeśli tak było istotnie, usuwamy cały słownik `Byly[Waga(`v_0`)].Korz` oraz wszystkie kolejne (co do wagi) słowniki, których liczniki `Ile` są zerami. Manewr ten zapewnia oszczędność pamięci. Nie można go wykonać od razu, gdy tylko licznik `Ile` spadnie do zera; musi być jeszcze spełniony warunek, że usuwany słownik ma największą wagę, gdyż w przypadku istnienia słownika z większą wagą moglibyśmy się narazić na sytuację, w której wylanie wody z jakiejś cięższej konfiguracji dałoby element, który powinien trafić do tego słownika. Przecież kolejność badania konfiguracji bynajmniej nie jest zgodna z ich wagami.

Odpowiednie fragmenty kodu definiujące omówione struktury danych znajdują się w modułach TYPY.PAS i KOLEJKA.PAS.

Program główny, organizujący przejście grafu wszerz, jest zawarty w pliku MOK.PAS i przedstawia się następująco:

```
begin {Poczatek programu MokraRobota z pliku MOK.PAS.}
 CzytajDane(IleNaczyn,sp,sk);
  { wczytuje liczbe naczyn, sp i sk z pliku MOK.IN }
 if Rowne(IleNaczyn,sp,sk)
  then WypiszWynik(0)
  else begin
   if AnalizaWstepna(IleNaczyn,sp,sk) then begin
    { jesli wstepnie wychodzi, ze mozna osiagnac sk z sp }
    JestRozw:=false;
    Inicjuj(IleNaczyn,sp,WagaKoncowa);
     { wstaw do kolejki stan poczatkowy z liczba krokow 0 }
    while not KolejkaPusta and not JestRozw do begin
     Pobierz(IleNaczyn,s,Krok);
      { pobierz pierwszy stan z kolejki }
     JestRozw:=Osiagalna(IleNaczyn,s,sk,Krok);
      { sprawdz czy sk osiagalna z s w jednym kroku. }
      { Wstaw do kolejki wszystkie osiagalne, ktore jeszcze
        nie byly rozwazane }
     UsunZbedne
      { Usun juz nieistotne informacje }
    end; {while}
    if JestRozw then WypiszWynik(Krok+1)
```

```
    else WypiszWynik(-1)
   end {if}
   else { Wstepna analiza odrzucila istnienie rozwiazan }
    WypiszWynik(-1);
    Sprzataj
  end {sp<>sk}
end. {MokraRobota}
```

Omówienie rozwiązań podanych przez uczniów

Głównym problemem w tym zadaniu było sprawdzanie, czy badana konfiguracja wystąpiła już dotychczas. Na pomysł zrobienia słownika na drzewie binarnych wyszukiwań wpadł jeden uczeń. Inni, którzy również otrzymali maksymalną liczbę punktów, poradzili sobie z problemem słownika zauważając następującą, wspomnianą już na początku własność. Choć wszystkich czwórek liczb z przedziału [0, 49] jest 50^4, czyli 6250000, co w standardowej pamięci nie mieści się nawet w postaci mapy bitowej, to ze względu na to, że w każdym stanie któreś z naczyń jest puste bądź pełne, liczba możliwych konfiguracji maleje dość istotnie. Wystarczy bowiem pamiętać które z 4 naczyń jest tym wyróżnionym (pustym bądź pełnym) oraz w jaki sposób. Potrzeba na to raptem 8 wartości (a nie 50). Pozostałe 3 naczynia dają 50^3 kombinacji, co przemnożone przez 8 daje zaledwie 1 000 000. Liczba różnych konfiguracji dla każdych danych wejściowych nie przekracza zatem miliona, co za pomocą mapy bitowej zapisuje się zaledwie w 125 000 bajtów. Taka tablica mieści w standardowej pamięci. Takie rozwiązanie podało 10 uczniów.

Mapa bitowa (czyli kompresja tablicy logicznej) jest oczywiście niezwykle szybka: w koszcie stałym, i to niedużym, pozwala stwierdzić, czy dany element występuje w strukturze, czy go nie ma. Również zaznaczanie elementu odbywa się bardzo szybko. Nadaje się więc do problemu słownika idealnie, jeśli przestrzeń danych spośród których tworzymy słownik jest na tyle mała, że cała mieści się w pamięci. Wadą tej metody jest usztywnienie rozmiaru pamięci, której przy nieco większych danych mogłoby po prostu zabraknąć.

Innymi spotykanymi rozwiązaniami było użycie listy, z jej przeglądaniem liniowym lub przeszukiwaniem binarnym, co spowalniało jednak algorytm.

Niektórzy sprawdzali wstępne warunki, aby stwierdzić, czy w ogóle jest szansa znalezienia rozwiązania. Nikt nie sprawdzał jednak opisanego na początku warunku z największym wspólnym dzielnikiem. Niektórzy tylko badali parzystość (czyli przypadek, gdy $\mathrm{NWD}(o_1,\ldots,o_n) = 2$). Kilku zawodników sprawdzało również na początku warunek, czy końcowa konfiguracja zawiera naczynie pełne lub naczynie puste.

Cenną optymalizacją było też wyhamowanie przetwarzania, gdy zchodziło z wagą poniżej wagi konfiguracji końcowej. Ci zawodnicy, którzy używali mapy bitowej nie musieli nawet tego robić, aby dostać maksymalną liczbę punktów; szybkość ich rozwiązania rekompensowała straty wynikające ze zbyt głębokiej analizy.

Do typowych błędów w rozwiązaniach tego zadania należało:

1. Użycie statycznej pamięci do reprezentowania grafu (bez mapy bitowej). Powodowało to problemy z pamięcią w przypadku dużych danych.
2. Zastosowanie rekurencji bez ucinania raz już przebadanych wierzchołków.
3. Zastosowanie rekurencji prowadzącej do innej strategii przechodzenia grafu niż wszerz. Pierwsze otrzymane rozwiązanie nie jest na ogół optymalne, więc albo trzeba zrobić pełne przeszukanie grafu, albo tylko przez przypadek można trafić na rozwiązanie właściwe.
4. Nieoszczędna gospodarka pamięcią (nie zwalnianie struktur nieprzydatnych). Dla największych testów stanowiło to problem nie do przejścia.

Testy

Testy, na których sprawdzane były rozwiązania, można podzielić na parę klas.

0. Test z treści zadania: MOK0.IN.
1. Testy poprawnościowe: MOK1.IN – MOK3.IN.
2. Testy brzegowe: MOK4.IN – MOK7.IN.
3. Testy wydajnościowe: MOK8.IN – MOK9.IN.
4. Test pamięciowy: MOK10.IN.

Poszczególne testy przedstawiały się następująco:

MOK0.IN – test z treści zadania.
MOK1.IN – test mający na celu wyeliminowanie rozwiązań przeszukujących graf w głąb.
MOK2.IN – test mający na celu wyeliminowanie rozwiązań przeszukujących graf w głąb oraz zbyt szybko usuwających przebadane stany.
MOK3.IN – test eliminujący rozwiązania usuwające przebadane stany.
MOK4.IN – test z identyczną konfiguracją początkową i końcową.
MOK5.IN – test wymagający wykonania tylko jednego kroku.
MOK6.IN – test z jednym naczyniem, bez rozwiązania.
MOK7.IN – test na to, czy badane są warunki wstępne (podzielność).
MOK8.IN – test odsiewający rozwiązania z nieoptymalnym słownikiem lub bez niego.
MOK9.IN – test odsiewający rozwiązania z nieoptymalnym szukaniem istniejących stanów.
MOK10.IN– test kontrolujący gospodarkę pamiecią.

7.1.4. Zadanie GOŃCY (Autor: Krzysztof Diks)

Treść zadania GOŃCY

Król Informaticus, władca Bajtocji, ma zmartwienie. Tajne służby doniosły mu, że zły król Haker postanowił napaść na jego królestwo. Co więcej, Haker wysłał już tajnych agentów do jednego z miast Bajtocji poza stolicą, by utrudniali królowi Informaticusowi przygotowania do obrony. Informaticus postanowił powiadomić mieszkańców kraju o grożącym niebezpieczeństwie.

Obywatele Bajtocji mieszkają tylko w miastach. Miasta są ponumerowane od 1 do n, gdzie $3 \leq n \leq 500$. Stolica kraju ma numer 1. W kraju istnieje dobrze rozwinięta sieć dróg. Wszystkie drogi są dwukierunkowe. Dowolne dwa miasta mają co najwyżej jedno bezpośrednie połączenie drogowe, ale dla każdych dwóch różnych miast A, B można przejechać z A do B, a następnie wrócić z B do A nie przejeżdżając ponownie przez żadne z miast na trasie z A do B.

Król Informaticus postanowił wysłać niezależnie dwóch gońców, by ostrzegli mieszkańców wszystkich miast o grożącym niebezpieczeństwie. Każdy goniec porusza się po drogach Bajtocji i może wielokrotnie odwiedzać to samo miasto. Jednak kolejność, w jakiej goniec odwiedza poszczególne miasta po raz pierwszy, musi być zgodna z planem, który otrzymał od króla.

Plan jest ciągiem n różnych liczb całkowitych $x_1, x_2, \ldots, x_n$ z zakresu $[1..n]$, przy czym $x_1 = 1$. Po drodze z miasta o numerze x_i do miasta o numerze x_{i+1} goniec może przejść przez dowolną, skończoną liczbę miast, ale tylko tych, które już wcześniej odwiedził. Niestety, na posłańców czyhają ludzie Hakera i biorą w niewolę każdego gońca, który trafi do opanowanego przez nich miasta. Trzeba przygotować takie plany tras podróży obu gońców, aby do każdego miasta Bajtocji dotarł przynajmniej jeden z nich, zanim obaj zostaną schwytani przez ludzi Hakera, ukrytych w jednym z miast poza stolicą.

Zadanie

Ułóż program, który:

- wczytuje z pliku tekstowego GON.IN liczbę n wszystkich miast w Bajtocji, liczbę d wszystkich odcinków dróg łączących bezpośrednio dwa różne miasta, a następnie opisy wszystkich bezpośrednich połączeń drogowych,
- układa plany tras obu gońców zaczynające się w stolicy w taki sposób, aby do każdego miasta w Bajtocji dotarł przynajmniej jeden z nich zanim obaj zostaną schwytani przez ludzi Hakera,
- zapisuje wyznaczone plany w pliku tekstowym GON.OUT.

Wejście

W pierwszym wierszu pliku tekstowego GON.IN jest zapisana jedna liczba naturalna n – jest to liczba wszystkich miast w Bajtocji.

W drugim wierszu jest zapisana jedna liczba naturalna d – jest to liczba wszystkich bezpośrednich połączeń drogowych.

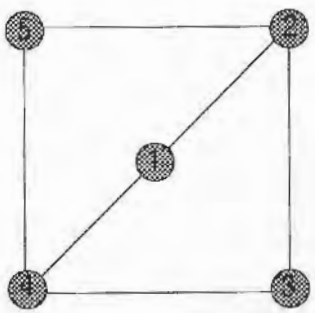
W każdym z kolejnych d wierszy znajduje się opis jednego bezpośredniego połączenia drogowego. Opis bezpośredniego połączenia to dwie różne liczby naturalne z zakresu $[1..n]$ oddzielone pojedynczym odstępem. Każde połączenie występuje w pliku tylko raz.

Wyjście

W pierwszym wierszu pliku tekstowego GON.OUT należy zapisać plan trasy pierwszego gońca w postaci ciągu n różnych liczb naturalnych z zakresu $[1..n]$ pooddzielanych pojedynczym odstępem.

W drugim wierszu należy zapisać w takiej samej postaci plan trasy drugiego gońca.

Przykład



Opisem przedstawionej na rysunku sieci dróg Bajtocji jest następujący plik GON.IN:

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

W tym przypadku poprawnym rozwiązaniem jest plik GON.OUT:

1 2 3 5 4
1 4 5 3 2

Pierwszy goniec mógłby wtedy poruszać się po drodze: 1 2 3 2 5 4, a drugi – po drodze: 1 4 5 4 3 2.

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

Rozwiązanie zadania GOŃCY

Rozważania rozpoczniemy od następującego spostrzeżenia:

Jeżeli $x_1 = 1, x_2, \ldots, x_n$ *jest planem trasy pierwszego gońca, to aby każde miasto zostało odwiedzone przez co najmniej jednego gońca, niezależnie od miejsca ukrycia nieprzyjaciela, plan trasy drugiego gońca musi mieć postać:* $x_1 = 1, x_n, x_{n-1}, \ldots, x_2$.

Powiemy, że plan $x_1, x_2, \ldots, x_n$ jest **bezpiecznym planem** wtedy i tylko wtedy, gdy spełnione są następujące cztery warunki:

1. Ciąg $x_1, x_2, \ldots, x_n$ jest ciągiem n różnych (miast) liczb naturalnych z przedziału $[1, 2, \ldots, n]$.
2. $x_1 = 1$.
3. Istnieje bezpośrednie połączenie drogowe pomiędzy (stolicą) x_1 i x_n.
4. Dla każdego $i = 2, \ldots, n-1$, miasto x_i ma bezpośrednie połączenia drogowe z miastami x_k, x_l, dla pewnych k, l takich, że $k < i < l$.

Łatwo teraz zauważyć, że w celu wyznaczenia planów tras gońców, które zagwarantują odwiedzenie każdego miasta przez co najmniej jednego z nich, niezależnie od miejsca ukrycia ludzi Hakera, wystarczy wyznaczyć jeden bezpieczny plan $x_1, x_2, \ldots, x_n$. Wówczas dobrymi planami tras gońców są: $x_1, x_2, \ldots, x_n$ i $x_1, x_n, \ldots, x_2$. Powstaje jednak pytanie, czy taki bezpieczny plan zawsze istnieje. Odpowiedź jest pozytywna. Przedstawimy algorytm konstruujący taki plan. Wprowadźmy najpierw kilka oznaczeń. Sieć połączeń drogowych w Bajtocji będziemy reprezentowali za pomocą grafu $G = (V, E)$, w którym zbiór wierzchołków V jest zbiorem miast $\{1, 2, \ldots, n\}$, a para xy jest krawędzią w E wtedy i tylko wtedy, gdy istnieje bezpośrednie połączenie drogowe pomiędzy x i y – o miastach x i y mówimy wtedy, że **sąsiadują ze sobą** (lub są **sąsiednie**). Dla każdego miasta x, oznaczmy przez $L(x)$ ustalony, ale dowolnie uporządkowany ciąg wszystkich miast z nim sąsiadujących. Ciąg $L(x)$ nazywamy **listą sąsiadów** x.

Możemy teraz podać opis algorytmu wyznaczania bezpiecznego planu.

Algorytm Bezpieczny Plan

begin
Niech z będzie pierwszym miastem na liście sąsiadów stolicy $L(1)$;
Bezpieczny := $(1, z)$; *Kolejny* := 1;
while *Kolejny* $\neq z$ **do**
if wśród sąsiadów miasta *Kolejny* jest takie miasto,
które nie występuje jeszcze w ciągu *Bezpieczny* **then**
znajdź ciąg parami różnych miast $m_1, m_2, \ldots, m_k$ taki, że
– $k > 2$,
– $m_1 =$ *Kolejny*, $m_k \neq m_1$ i m_k jest elementem ciągu *Bezpieczny*,
– dla każdego $i = 2, \ldots, k-1$, m_i nie występuje w ciągu
Bezpieczny i sąsiaduje z miastami m_{i-1} oraz m_{i+1};
rozszerz ciąg *Bezpieczny* wstawiając do niego ciąg
$m_2, m_3, \ldots, m_k - 1$ bezpośrednio po elemencie *Kolejny*
else
rozważ *Kolejny* element ciągu *Bezpieczny*;
przekaż jako wynik ciąg *Bezpieczny*
end

Pokażemy teraz, że ten algorytm rzeczywiście wyznacza bezpieczny plan. W tym celu wystarczy zauważyć, że następujące warunki są niezmiennikami pętli **while**:

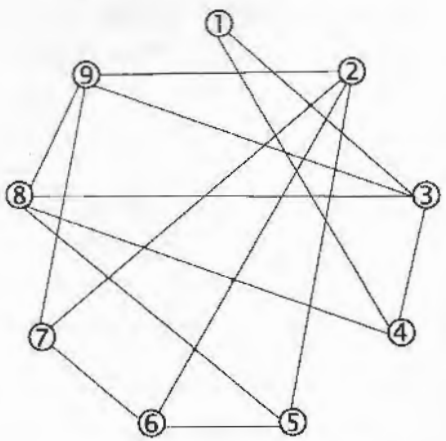
1. Ciąg *Bezpieczny* jest ciągiem różnych liczb całkowitych (miast) z przedziału $[1, n]$.
2. Pierwszym elementem ciągu *Bezpieczny* jest (stolica) 1.
3. Istnieje bezpośrednie połączenie drogowe pomiędzy stolicą a ostatnim miastem (z) w ciągu *Bezpieczny*.
4. Dla każdego miasta x z ciągu *Bezpieczny*, różnego od pierwszego i ostatniego miasta z tego ciągu, istnieją bezpośrednie połączenia drogowe z miastami v i w takimi, że v, w występują w ciągu *Bezpieczny*, v poprzedza x, a w występuje po x w tym ciągu.
5. Każde miasto sąsiadujące z jakimkolwiek miastem poprzedzającym miasto *Kolejny* w ciągu *Bezpieczny* znajduje się już w tym ciągu.

Z tych warunków bezpośrednio wynika, że jeżeli *Kolejny* jest równe z (ostatniemu miastu w ciągu *Bezpieczny*), to wynikowy ciąg jest rzeczywiście planem bezpiecznym. Aby wykazać, że powyższe warunki są rzeczywiście niezmiennicze względem pętli **while** zauważmy, że jeśli wśród sąsiadów miasta *Kolejny* jest

taki, który nie występuje jeszcze w ciągu *Bezpieczny*, to ciąg $m_1, m_2, \ldots, m_k$ rzeczywiście istnieje. Z treści zadania wiemy, że dla każdych dwóch różnych miast A, B w Bajtocji można przejechać z A do B, a następnie wrócić z B do A, nie przejeżdżając ponownie przez żadne z miast znajdujące się na trasie z A do B. Stąd wynika, że jeżeli v jest sąsiadem *Kolejny* nie należącym do ciągu *Bezpieczny*, to istnieje trasa z v do ostatniego miasta w ciągu *Bezpieczny* (do miasta z) nie przechodząca przez *Kolejny*. Niech $u_1 = v, u_2, \ldots, u_l = z$ będzie taką trasą i niech u_i będzie pierwszym miastem na tej trasie należącym do ciągu *Bezpieczny*. Z warunku 5 wynika, że u_i występuje w ciągu *Bezpieczny* po *Kolejny*. Zatem poszukiwanym ciągiem może być ciąg $m_1 = Kolejny$, $m_2 = u_1$, $m_3 = u_2$, $\ldots$, $m_k = u_i$, dla $k = i + 1$. Udowodniliśmy w ten sposób, że podany przez nas algorytm poprawnie znajduje bezpieczny plan.

Opiszemy teraz realizację algorytmu **Bezpieczny Plan** w programie GON.PAS. Graf połączeń jest pamiętany w postaci list sąsiedztwa. Sąsiadów każdego miasta `i` pamiętamy w jednokierunkowej liście o początku wskazywanym przez `L[i]`. Listy sąsiedztwa można zbudować w czasie liniowym $O(n + d)$ – wykonuje to procedura `Inicjalizacja`.

Szukany bezpieczny plan jest dwukierunkową listą o początku wskazywanym przez `bPocz` i końcu wskazywanym przez `bOst`. Dalej, dla prostoty, tę listę nazywamy *Bezpieczny*. Bezpieczny plan jest inicjowany w taki sposób, że jego pierwszym elementem jest stolica, a drugim i ostatnim w tym momencie – pierwsze miasto na liście sąsiadów stolicy. Bezpieczny plan jest tworzony w procedurze `UtworzBezpiecznyPlan`. Lista *Bezpieczny* zmienia się dynamicznie – jej przetwarzany element jest wskazywany przez `Kolejny`. Dla miasta wskazywanego przez `Kolejny` jest przeglądana jego lista sąsiadów. Po napotkaniu na



Rysunek 1. Przykładowy graf.

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

Rysunek. 2. Kolejno tworzone przez algorytm bezpieczne plany dla grafu z rys. 1.

niej miasta, które nie znajduje się na liście *Bezpieczny*, zostaje wywołana procedura `RozszerzPlan`, której zadaniem jest wyznaczenie ciągu miast, kończącego się miastem występującym na liście *Bezpieczny* po mieście wskazywanym aktualnie przez `Kolejny` – znaleziony ciąg jest umieszczany na liście *Bezpieczny* bezpośrednio po elemencie wskazywanym przez `Kolejny`. Wyznaczenie tego ciągu miast polega na znalezieniu odpowiedniej drogi parami różnych wierzchołków w grafie G, która prowadzi przez miasta nie będące jeszcze elementami listy *Bezpieczny*, zaczyna się w mieście sąsiadującym z miastem wskazywanym przez `Kolejny`, a kończy się w mieście sąsiadującym z miastem występującym na liście *Bezpieczny* po elemencie wskazywanym przez `Kolejny`. Na rysunku 2 są przedstawione kolejne postacie bezpiecznego planu, wyznaczanego dla grafu z rysunku 1.

Drogi rozszerzające bezpieczny plan są wyznaczane metodą przeszukiwania grafu w głąb. Status poszczególnych wierzchołków grafu w trakcie przeszukiwania jest pamiętywany w tablicy `Status`. Wierzchołek grafu (miasto) może znajdować się już na liście *Bezpieczny* – ma wtedy status `NaPlanie`, mógł zostać odwiedzony w trakcie wyznaczania drogi – ma status `Odwiedzony`, lub nie był jeszcze odwiedzony w takcie wyznaczania drogi – ma wtedy status `NieOdwiedzony`. Wyznaczanie drogi zabiera czas $O(n+d)$, gdzie n jest liczbą miast w Bajtocji, a d jest liczbą bezpośrednich połączeń między miastami. Ponieważ po znalezieniu drogi lista *Bezpieczny* jest rozszerzana o co najmniej jedno miasto i każde miasto z tej listy jest przeglądane tylko raz, łączny czas wykonania algorytmu **Bezpieczny Plan** wynosi $O(n(n+d))$. Faktycznie, przedstawiony algorytm działa bardzo szybko, ponieważ drogi, o które jest rozszerzana lista *Bezpieczny*, zawierają zazwyczaj więcej niż jeden wierzchołek.

Omówienie rozwiązań podanych przez uczniów

Spora część uczniów rozwiazała to zadanie bez trudu. Podane rozwiązania można podzielić na 3 grupy:

1. Rozwiązania podobne do rozwiązania przedstawionego powyżej.
2. Rozwiązania, w których obliczenia rozpoczynają się od dowolnego planu trasy dla pierwszego gońca, a następnie ten plan jest modyfikowany (przez zamianę miast) w taki sposób, aby uzyskać plan bezpieczny.
3. Rozwiązania oparte na przeszukiwaniu z nawrotami.

Testy

Do sprawdzania rozwiązań użyto 12 testów GON0.IN, GON1.IN, ..., GON11.IN. Dziewięć testów miało umiarkowane rozmiary, od 3 do 500 miast i od 3 do 3000 bezpośrednich połączeń. Ich celem było przetestowanie, jak programy radzą sobie z wyszukiwaniem dróg w grafie połączeń, które umożliwiają rozszerzenie dynamicznie budowanego bezpiecznego planu. Testy te były tak dobrane, aby podczas budowania drogi przeszukiwanie przenosiło się do „ślepych zaułków", tzn. do miast, z których nie można rozszerzyć drogi i należy wykonać powrót do wcześniejszych wierzchołków na drodze.

Zadaniem trzech testów było sprawdzenie szybkości działania ocenianych programów.

7.2. Zawody II stopnia

7.2.1. Zadanie PIĘĆ MONET (Autor: Piotr Chrząstowski-Wachtel)

Treść zadania PIĘĆ MONET

Mamy pięć monet A, B, C, D, E, których ciężary są parami różne. Dysponując bardzo czułą wagą szalkową, pozwalającą porównać ze sobą ciężary dwóch monet, uszereguj je od najlżejszej do najcięższej. Waga jest delikatnym i kosztownym urządzeniem. Pierwszych siedem porównań możemy wykonać za darmo. Ósme kosztuje 1 złoty, a każde następne porównanie jest 125 razy droższe od poprzedniego. Decyzję o tym, które porównanie wykonać w danej chwili, możemy podejmować w zależności od wyników poprzednich pomiarów. Zaprojektuj algorytm sortujący monety, tak aby koszt ważenia był jak najmniejszy.

Zadanie

Rozwiązanie możesz zapisać w Pascalu lub w języku C.

Pascal

Na dyskietce otrzymasz program, który pozwoli Ci uruchomić Twoje rozwiązanie. Uzupełnij treść procedury *sortujMonety* z pliku USORT.PAS. Ta pro-

cedura powinna, korzystając z funkcji *lzejsza*, uszeregować monety według ich ciężaru od najlżejszej do najcięższej, a następnie, wywołując procedurę *wynik*, przekazać znalezioną kolejność. Ocena poprawności rozwiązania dla pewnych dowolnie wybranych ciężarów monet zostanie dokonana na podstawie tego wywołania procedury *wynik*.

Kod źródłowy programu jest zapisany w następujących plikach:

UMONETA.PAS	– definicja typu *Tmoneta*,
UWAGA.PAS	– funkcja *lzejsza* i procedura *wynik*,
USORT.PAS	– miejsce na Twoje rozwiązanie,
MONETY.PAS	– główna część programu.

Procedura *sortujMonety* powinna korzystać z typu *TMoneta*, zdefiniowanego w pliku UMONETA.PAS

```
unit UMoneta;
interface
 type
  TMoneta = (A, B, C, D, E);
 {Z pozostałych deklaracji, zawartych w tym pliku, korzystać nie wolno.}
implementation
 {...}
```

Do porównywania monet oraz podawania rozwiązania służą funkcja *lzejsza* i procedura *wynik* zadeklarowane w pliku UWAGA.PAS:

```
unit UWaga;
 interface
 uses UMoneta;
 function lzejsza(m1,m2:TMoneta):boolean;
   {Przyjmuje wartość true jeśli moneta m1 jest lżejsza od monety m2,
    w przeciwnym przypadku przyjmuje wartość false.}
 procedure wynik(m1,m2,m3,m4,m5:TMoneta);
   {Gdy monety zostaną już uszeregowane od najlżejszej do najcięższej, należy
    wywołać procedurę wynik. Parametry m1, m2, m3, m4, m5 oznaczają
    monety uszeregowane od najlżejszej do najcięższej. Zwróć uwagę, że ta
    procedura musi być wywołana dokładnie jeden raz w trakcie działania
    sortujMonety.}
implementation
 {...}
```

Rozwiązanie zapisz w pliku USORT.PAS:

```
unit USort;
interface
  uses UMoneta,UWaga;
  procedure sortujMonety;
implementation
 {Poniżej jest miejsce na kod Twojego rozwiązania. Typy, zmienne, funkcje oraz
   procedury, które zadeklarujesz muszą być lokalne dla tego modułu.}
  procedure sortujMonety;
  begin
  end;
```

Spośród funkcji, procedur, typów oraz zmiennych zadeklarowanych w innych plikach wolno Ci korzystać tylko z funkcji *lzejsza*, procedury *wynik* i typu *TMoneta*. Nie wolno zmieniać czegokolwiek poza implementacją modułu USORT.PAS. Ocenie podlega tylko plik USORT.PAS, który zostanie skompilowany z innym programem sprawdzającym poprawność rozwiązania, dlatego interfejs modułu musi pozostać nie zmieniony.

Język C

Na dyskietce otrzymasz program, który pozwoli Ci uruchomić Twoje rozwiązanie. Uzupełnij treść procedury *sortujMonety* z pliku USORT.C. Ta procedura powinna, korzystając z funkcji *lzejsza*, uszeregować monety według ich ciężaru od najlżejszej do najcięższej, a następnie, wywołując procedurę *wynik*, przekazać znalezioną kolejność. Ocena poprawności rozwiązania dla pewnych dowolnie wybranych ciężarów monet zostanie dokonana na podstawie tego wywołania procedury *wynik*.

Kod źródłowy programu jest zapisany w następujących plikach:

UMONETA.H	UMONETA.C	– definicja typu *TMoneta*,
UWAGA.H	UWAGA.C	– funkcja *lzejsza* i procedura *wynik*,
USORT.H	USORT.C	– miejsce na Twoje rozwiązanie,
MONETY.C		– główna część programu

Procedura *sortujMonety* powinna korzystać z typu *TMoneta*, zdefiniowanego w pliku UMONETA.H:

```
typedef enum {A, B, C, D, E} TMoneta;
/* Z pozostałych deklaracji, zawartych w tym pliku, korzystać nie wolno. */
```

Do porównywania monet oraz podawania rozwiązania służą funkcja *lzejsza* i procedura *wynik* zadeklarowane w pliku UWAGA.H:

```
extern int lzejsza (TMoneta m1, TMoneta m2);
/* Przyjmuje wartość 1 (prawda) jeśli moneta m1 jest lżejsza od monety m2,
   w przeciwnym przypadku przyjmuje wartość 0 (fałsz).*/
extern void wynik (TMoneta m1, TMoneta m2, TMoneta m3, TMoneta m4,
                   TMoneta m5);
/* Gdy monety zostaną już uszeregowane od najlżejszej do najcięższej, należy
   wywołać procedurę wynik. Parametry m1, m2, m3, m4, m5 oznaczają monety
   uszeregowane od najlżejszej do najcięższej. Zwróć uwagę, że ta procedura
   musi być wywołana dokładnie jeden raz w trakcie działania sortujMonety.*/
```

Rozwiązanie zapisz w pliku USORT.C:

```
#include "usort.h"
/* Tu mogą być umieszczone zmienne i funkcje pomocnicze */
void sortujMonety()
{
/* To jest miejsce na twoje rozwiązanie */
/* Można wywołać tylko funkcje lzejsza(...) i wynik(...) oraz własne funkcje
   z tego modułu. */
}
```

Spośród funkcji, procedur, typów oraz zmiennych zadeklarowanych w innych plikach wolno Ci korzystać tylko z funkcji *lzejsza*, procedury *wynik* i typu *TMoneta*. Nie wolno również zmieniać czegokolwiek poza implementacją modułu USORT.C. Ocenie podlega tylko plik USORT.C, który zostanie skompilowany z innym programem sprawdzającym poprawność rozwiązania, dlatego interfejs modułu musi pozostać nie zmieniony.

Rozwiązanie zadania PIĘĆ MONET

Wszystkich możliwych do zadania pytań jest $\binom{5}{2} = 10$. Nie wszystkie jednak musimy zadać, gdyż pewne nierówności mogą wynikać z wcześniejszych odpowiedzi. Stanie się tak, jeżeli na przykład A < B < C, nic nie wiemy o związku D z A, B, C i zadamy pytanie o to, czy D jest mniejsze od B. Niezależnie od odpowiedzi otrzymamy jedno porównanie „za darmo": jeśli D < B, to mamy również D < C, a jeśli B < D, to również A < D. Ponieważ zadanie każdego pytania powyżej siódmego kosztuje, więc zminimalizowanie liczby porównań jest kluczowe dla tego zadania, szczególnie, że być może można je zrobić za darmo.

Aby tak się stało nie możemy wykonać więcej niż 7 porównań. W tym celu trzeba zaoszczędzić na 3 odpowiedziach, czyli wyniki 3 porównań wydedukować z pozostałych i to nawet przy najbardziej niesprzyjających odpowiedziach. Czy

jest to w ogóle możliwe? Zauważmy, że odpowiedź na każde poprawnie zadane pytanie powoduje, że liczba możliwych porządków spełniających dotychczasowe odpowiedzi maleje. Z początku maleje nawet o połowę. Przypuśćmy, że zadaliśmy pytanie, czy A < B? Spośród 120 permutacji 5 elementów, dokładnie 60 spełnia A < B i 60 spełnia A > B. O ile przed zadaniem tego pytania mieliśmy 120 możliwych uporządkowań, to po uzyskaniu odpowiedzi na to pytanie będziemy mieli ich już tylko 60. Powiedzmy, że kolejnym pytaniem będzie, czy C < D? I znów, niezależnie od wariantu odpowiedzi na pierwsze pytanie, 30 spośród pozostałych permautacji spełni nierówność C < D, a pozostałe 30 – nierówność D < C. Gdyby za każdym razem udawało się nam mniej więcej o połowę zmniejszać liczbę możliwych wariantów, to moglibyśmy się spodziewać zredukowania możliwych układów do jednego (czyli po prostu ustalenia porządku między wszystkimi elementami) po $\lceil \log_2 120 \rceil = 7$ krokach. Teoretycznie jest więc to możliwe. Nie wiemy jeszcze jednak, czy tak równomierne obniżanie liczebności zbioru układów jest możliwe. Zauważmy, że różnica między $5! = 120$, a $2^7 = 128$ jest niewielka, więc na żadną rozrzutność nie możemy sobie pozwolić. Rozrzutnością nazwiemy takie pytanie, które w przypadku szczęścia pozwala na szybkie osiągnięcie wyniku (czyli gwałtowne zredukowanie liczby możliwych układów), ale przy pechu niewiele zmniejszy liczbę możliwych układów.

Pierwsze podejście jest standardowe: umieszczamy binarnie każdy następny element do poprzednio posortowanych, poczynając od jednoelementowego ciągu A (posortowanego). Za pomocą jednego porównania ustawiamy A i B. Powiedzmy, że A < B (jeżeli wyjdzie inny wynik porównania, to zamienimy rolami A z B). Następnie porównujemy C na przykład z A. Jeżeli mamy szczęście, to C jest mniejsze niż A. Wtedy nie musimy robić dodatkowego porównania. Ale gdy C jest większe niż A, wówczas musimy porównać je jeszcze z B. Będziemy więc potrzebowali w najgorszym razie łącznie 3 porównań.

Czwarty element umieszczamy za pomocą dwóch dodatkowych porównań. Porównujemy go najpierw ze środkowym z uszeregowanych już trzech elementów, a następnie, w zależności od wyniku jeszcze tylko z jednym: albo z najmniejszym z nich, jeśli C było mniejsze, albo z największym z nich, jeśli C było większe od elementu środkowego. Zużyliśmy już 5 porównań, aby uporządkować 4 elementy (szybciej czterech elementów uporządkować nie można, gdyż $4! = 24$, a $\lceil \log_2 24 \rceil = 5$).

Piąty element musimy porównać z drugim lub trzecim co do wielkości. Jeżeli będziemy mieli znowu pecha, to trzeba będzie jeszcze wykonać dwa porównania, razem 8 – za dużo. Warto zauważyć, że nie jest to zły wynik: 8 porównań będziemy mieli tylko przy dwukrotnym pechu: zarówno przy umieszczaniu trzeciego, jak i piątego elementu.

Istnieje jednak metoda umożliwiająca nawet w najgorszym przypadku wykonanie tylko 7 porównań. W tym miejscu gorąco zachęcamy Czytelników, aby przerwali czytanie i sami spróbowali zmierzyć się z tym problemem.

Wiemy już, że aby uzyskać w najgorszym przypadku 7 porównań, musimy wymusić uzyskanie trzech odpowiedzi za darmo, jako wniosków z dotychczas udzielonych. W tym celu staramy się utworzyć możliwie dużo uporządkowanych trójek, tak aby w efekcie porównania z elementem środkowym wydedukować dodatkową relację. Nasz algorytm dzielimy na dwie fazy.

Faza pierwsza. Porównujemy A z B, C z D. Załóżmy, że zachodzą nierówności $A < B$ i $C < D$ (jeżeli jest inaczej, to zamieniamy rolami odpowiednie elementy). W trzecim porównaniu porównujemy A z C; załóżmy, że $A < C$ (sytuacja cały czas jest symetryczna: jeżeli wyjdzie wynik odwrotny, to zamienimy A i C rolami). W tym momencie otrzymaliśmy jedną odpowiedź za darmo: wiemy, że $A < D$.

Faza druga. Porównujemy C z E.

Przypadek $E < C$. Dostajemy wtedy drugą nierówność za darmo: $E < D$. Porównując teraz B z C dostaniemy za darmo trzecią nierówność: jeżeli $B < C$ to $B < D$, a jeśli $C < B$, to $E < B$.

Przypadek $C < E$. Dostajemy drugą nierówność za darmo: $A < E$. W piątym porównaniu porównujemy D z E. Mamy tu dwa przypadki:

a. jeśli $D < E$, to porównujemy B z D i w obu przypadkach dostajemy za darmo trzecią nierówność: jeśli $B < D$, to $B < E$, a jeśli $D < B$, to $C < B$.
b. jeśli $E < D$, to porównujemy B z E i tak jak w poprzednim przypadku dostajemy za darmo trzecią nierówność: jeśli $B < E$, to $B < D$, a jeśli $E < B$, to $C < B$.

Rzecz jasna, gdy mamy już trzy nierówności za darmo, to resztę pytań zadajemy po prostu po kolei. Na pewno nie przekroczymy wtedy łącznie siedmiu porównań.

Na zakończenie przyjrzyjmy się sortowaniom krótkich ciągów z bliska. Wiadomo już, że dla ciągu pustego i jednoelementowego trzeba wykonać 0 porównań. Ciąg dwuelementowy potrzebuje jednego prównania, trzyelementowy – 3, czteroelementowy – 5, pięcioelementowy – 7. Jak to wygląda dalej? Czy zawsze minimalna liczba porównań potrzebnych do posortowania n elementów wynosi $\lceil \log_2 n! \rceil$? Okazuje się, że nie. Wzór ten jest jedynie dolnym ograniczeniem (zob. [11]), które jest przyjmowane dla $n < 12$ i dla pewnych większych wartości n, np. $n = 19$, $n = 20$. Wiadomo jednak, że już np. dla $n = 12$ potrzeba przynajmniej 30 porównań, choć $\lceil \log_2 12! \rceil = 29$.

Warto dodać, że zazwyczaj standardowe metody sortowania (takie jak quicksort, sortowanie przez kopcowanie, czy przez proste wstawianie) nie są op-

tymalne w przypadku małych danych. Takie przypadki często występują w praktyce, jako część innych algorytmów. Dla przykładu sortowanie 5 elementów jest częścią liniowego algorytmu znajdowania mediany zbioru, czyli wartości, dla której połowa elementów zbioru jest niemniejsza i połowa jest niewiększa.

Omówienie rozwiązań podanych przez uczniów

Żaden z uczniów nie rozwiązał tego zadania przy zerowym koszcie. Część zatrzymała się na 8 porównaniach, ale byli i tacy, którzy wykonali pełne 10 porównań, a nawet więcej. Trzeba przyznać, że zadanie to jest trudne. Autor tego zadania, w wieku uczestników Olimpiady, myślał nad rozwiązaniem parę tygodni, nie wiedząc, czy takie rozwiązanie istnieje.

Testy

Tym razem zawodnicy mieli napisać część programu: jedną procedurę, która wklejona do programu testującego umożliwiała konstruowanie złośliwych odpowiedzi, w zależności od sytuacji. Ponieważ dane do zadania były niewielkie, więc każdy program został przetestowany dla wszystkich 120 permutacji. Po prostu jeden test polegał na podawaniu odpowiedzi zgodnie z jednym ze 120 porządków. W wyniku dostawaliśmy cenę, jaką algorytm musiałby zapłacić za komplet sytuacji. Rekordziści płacili 2 zł, czyli tylko w 2 przypadkach potrzebowali 8 porównań.

7.2.2. Zadanie WIEŻE (Autor: Krzysztof Diks)

Treść zadania WIEŻE

Na szachownicy $n \times n$ ustawiamy n wież, gdzie $1 \le n \le 3000$. Ustawienie wież musi spełniać następujące warunki.

- Dla każdego $i = 1, \ldots, n$, i-tą wieżę wolno postawić tylko w prostokącie określonym przez dwie pary współrzędnych: (a_i, b_i), (c_i, d_i), gdzie (a_i, b_i) to współrzędne pola w lewym górnym rogu prostokąta (wiersz, kolumna), zaś (c_i, d_i) to współrzędne pola w prawym dolnym rogu, $1 \le a_i \le c_i \le n$ oraz $1 \le b_i \le d_i \le n$. Pole w lewym górnym rogu szachownicy ma współrzędne (1, 1), zaś pole w prawym dolnym rogu ma współrzędne (n, n).
- Żadne dwie wieże nie mogą się atakować, to znaczy nie mogą stać w tym samym wierszu lub tej samej kolumnie.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego WIE.IN rozmiar szachownicy n oraz dla każdego $i = 1, \ldots, n$ współrzędne prostokąta, w którym wolno postawić i-tą wieżę,
- bada, czy można ustawić wieże, każdą w ustalonym dla niej prostokącie, tak aby żadne dwie nie atakowały się wzajemnie i jeśli tak, znajduje jedno takie ustawienie,
- zapisuje w pliku tekstowym WIE.OUT jedno ustawienie wież spełniające warunki zadania albo słowo NIE, jeśli takie ustawienie nie istnieje.

Wejście

W pierwszym wierszu pliku WIE.IN jest zapisana jedna liczba całkowita dodatnia $n \leq 3000$.

W każdym z n następnych wierszy są zapisane cztery liczby całkowite dodatnie nie większe niż n, oddzielone pojedynczym odstępem. Liczby w i-tym z tych wierszy są współrzędnymi prostokąta, w którym wolno postawić i-tą wieżę.

Wyjście

W pliku WIE.OUT należy zapisać:
jedno słowo NIE,
albo w każdym z kolejnych n wierszy dwie liczby całkowite oddzielone pojedynczym odstępem. Liczby w i-tym wierszu powinny określać ustawienie i-tej wieży (wiersz kolumna); ta wieża powinna leżeć w prostokącie, którego współrzędne podano w $(i+1)$-szym wierszu pliku WIE.IN. Zwróć uwagę, że pozycje wież powinny być wypisane w takiej kolejności, w jakiej zostały wczytane współrzędne prostokątów, w których te wieże mogą się znaleźć.

Przykład

Dla pliku WIE.IN:

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

jednym z poprawnych rozwiązań jest plik WIE.OUT:

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

Spostrzeżenie 2. *Niech* $[p_s, k_s]$ *będzie najkrótszym przedziałem o początku* $p_s = 1$ *wśród danych przedziałów* $[p_i, k_i]$, $i = 1, \ldots, n$. *Wówczas, jeśli istnieje ustawienie dla tego zbioru przedziałów, to istnieje ustawienie w którym* $u_s = 1$. *(Zauważmy, że jeśli takiego przedziału nie ma, to dla tego zbioru przedziałów ustawienie nie istnieje.)*

Prawdziwość tego spostrzeżenia można uzasadnić następująco. Rozważmy ustawienie, w którym 1 nie jest reprezentantem najkrótszego przedziału o początku w 1. Niech $[p_s = 1, k_s]$ będzie najkrótszym przedziałem i niech u_s będzie jego reprezentantem. Niech $[p_l = 1, k_l]$ będzie przedziałem, którego reprezentantem jest 1. Ponieważ $u_s \leq k_l$, więc zamieniając reprezentatów tych przedziałów otrzymamy również poprawne ustawienie.

Korzystając ze spostrzeżenia 2 można znaleźć ustawienie (lub stwierdzić, że żadne ustawienie nie istnieje) dla zbioru przedziałów $P = \{ [p_i, k_i] : i = 1, 2, \ldots, n\}$ w następujący sposób. Znajdujemy najkrótszy przedział o początku w 1 i za reprezentanta tego przedziału w konstruowanym ustawieniu przyjmujemy 1. Następnie redukujemy nasze zadanie do zadania o $n - 1$ przedziałach zawartych w przedziale $[2, n]$. W tym celu z P usuwamy znaleziony odcinek, a wszystkie odcinki zaczynające się w 1 skracamy tak, aby zaczynały się w 2. Teraz, w taki sam sposób jak dla 1, wyznaczamy przedział, którego reprezentantem będzie 2. Kontynuujemy to postępowanie, aż znajdziemy reprezentantów dla wszystkich przedziałów lub stwierdzimy, że dla danego zbioru przedziałów ustawienia nie da się skonstruować. Procedura *Ustaw*, zamieszczona poniżej, jest bardziej szczegółowym opisem tego algorytmu. Kluczową rolę odgrywa w niej zbiór Q. Jeżeli poszukujemy przedziału, dla którego reprezentantem ma być j (j-ta iteracja pętli **while**), to Q zawiera wszystkie przedziały bez reprezentantów o początkach nie większych od j i końcach nie mniejszych niż j. O tych przedziałach można założyć, że zaczynają się w j. Wówczas, najkrótszym z nich jest ten, który ma najmniejszy koniec.

Kiedy możemy stwierdzić, że ustawienie dla odcinków z P nie istnieje? Po pierwsze wtedy, gdy nie możemy wskazać przedziału, dla którego j może być reprezentantem (zbiór Q jest pusty). Po drugie, gdy stwierdzimy, że po wybraniu przedziału, dla którego reprezentantem jest j, istnieje jeszcze przedział o końcu w j, dla którego dotychczas nie wybraliśmy reprezentanta. Ponieważ żaden z elementów $j + 1, \ldots, n$ nie może być reprezentantem tego przedziału, ustawienia nie można skonstruować.

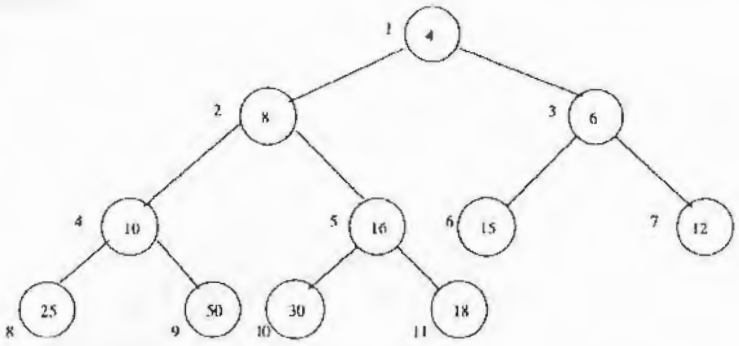
```
procedure Ustaw;
begin
  JestUstawienie := true; j := 0; Q := ∅;
  while JestUstawienie and (j < n) do begin
    j:= j + 1;
    Q := { [p_i, k_i] ∈ P : p_i = j } ∪ Q;                    { (1) }
     {Q jest zbiorem wszystkich przedziałów, dla których nie mamy jeszcze
      ustawienia i których początki są nie większe od j. }
    if Q = ∅ then JestUstawienie := false
    else begin
      znajdź w Q odcinek [p_i, k_i] o najmniejszym końcu k_i;  { (2) }
      u_i := j;
      Q := Q - { [p_i, k_i] };                                 { (3) }
      JestUstawienie := Q nie zawiera przedziału o końcu j     { (4) }
    end
  end
end; {Ustaw}
```

Procedurę *Ustaw* można łatwo zrealizować tak, aby działała w czasie $O(n^2)$. Pokażemy teraz, jak należy ją zrealizować, by działała w czasie $O(n \log n)$ i ta jej realizacja jest zawarta w programie WIE.PAS i nosi taką samą nazwę `Ustaw`. Kluczowym elementem dla tej realizacji jest sposób reprezentowania zbioru Q. Zauważmy, że na zbiorze Q są wykonywane następujące operacje:

- dodawanie nowych przedziałów do Q (w wierszu oznaczonym przez (1));
- znajdowanie w Q przedziału z najmniejszym końcem (wiersze (2) i (4));
- usuwanie z Q przedziału z najmniejszym końcem (wiersz (3)).

Struktura danych, na której można wykonywać te trzy operacje, czyli dodawanie elementów, znajdowanie i usuwanie minimalnego elementu, nazywa się **kolejką priorytetową**. Istnieje wiele implementacji kolejek priorytetowych. Jedną z najprostszych, a jednocześnie – najefektywniejszych, jest **kopiec**, zob. [6]. Przedstawimy teraz reprezentację zbioru Q w postaci kopca.

Kopiec n-elementowy jest drzewem binarnym o n węzłach, w którym wszystkie poziomy, z wyjątkiem być może ostatniego, są całkowicie wypełnione, a poziom ostatni jest wypełniony od lewej do prawej strony. (**Uwaga.** W grafach, które reprezentują struktury danych, wierzchołki nazywamy **węzłami**.) Wysokość kopca wynosi $O(\lfloor \log n \rfloor)$. Jeżeli ponumerujemy węzły kopca kolejno od korzenia do liści, poziomami od lewej do prawej strony, to synami i-tego węzła w kopcu są węzły $2i$ oraz $2i + 1$ (o ile oczywiście istnieją), a ojcem i-tego węzła, dla $i > 1$, jest węzeł o numerze $\lfloor i/2 \rfloor$. Dzięki tym zależnościom kopiec można



Rysunek 1. Przykład kopca.

reprezentować w tablicy, w której i-ty element odpowiada i-temu węzłowi kopca. Trzy wymienione operacje można efektywnie wykonywać na kopcu dzięki rozmieszczeniu elementów zbioru w węzłach kopca w specjalnym porządku, zwanym **porządkiem kopcowym**: element umieszczony w i-tym węźle jest nie większy od elementów umieszczonych w jego synach $2i$ oraz $2i + 1$ (o ile synowie istnieją). Gwarantuje to, że minimalny element w kopcu znajduje się zawsze w jego korzeniu, czyli w węźle o numerze 1 (zob. rys. 1).

Zatem, element minimalny można znaleźć w kopcu w czasie stałym. Wstawianie elementu do kopca odbywa się w następujący sposób. Jeśli kopiec ma r elementów, to dla nowego elementu tworzymy nowy węzeł o numerze $r + 1$ i wstawiamy go do tego węzła. Wstawienie nowego elementu może zaburzyć porządek kopcowy, gdy element w węźle $\lfloor (r + 1)/2 \rfloor$ jest większy od niego. Jeśli tak, to należy zamienić elemnty w węzłach $r + 1$ i $\lfloor (r + 1)/2 \rfloor$, a następnie sprawdzić, czy nie jest zaburzony porządek kopcowy w węzłach $\lfloor (r + 1)/2 \rfloor$ i $\lfloor \lfloor (r + 1)/2 \rfloor /2 \rfloor$. Jeżeli porządek kopcowy jest ponownie zaburzony, zamieniamy elementy w tych węzłach i posuwamy się tak w kierunku korzenia, aż porządek kopcowy zostanie przywrócony. Nastąpi to najpóźniej z chwilą osiągnięcia korzenia (węzła o numerze 1). W programie WIE.PAS wstawianie nowego przedziału do kopca wykonuje procedura `WstawPrzedzial`. Przedziały w kopcu są uporządkowane względem swoich końców. Czas wstawiania do kopca wynosi $O(\log r)$.

Operacja usuwania elementu minimalnego z kopca jest operacją odwrotną do wstawiania. Jeżeli kopiec zawiera r elementów, to w miejsce usuwanego elementu minimalnego (z węzła o numerze 1) jest wstawiany element z węzła o numerze r, a następnie węzeł ten jest usuwany z kopca (r jest zmniejszane o 1). Umieszczenie nowego elementu w węźle 1 może ponownie zaburzyć po-

rządek kopcowy. Dzieje się tak wtedy, gdy któryś z elementów z węzłów 2 i 3 jest od niego mniejszy. Aby przywrócić porządek kopcowy, zamieniamy mniejszy z nich z elementem z węzła 1. Załóżmy, że element z korzenia powędrował do węzła 2. Wówczas, jeżeli żaden z elementów w węzłach 4 i 5 (synowie węzła 2) nie jest mniejszy od elementu z węzła 2, to porządek kopcowy jest przywrócony. W przeciwnym razie zamieniamy element z węzła 2 z mniejszym z elementów z węzłów 4 i 5. Kontynujemy to postępowanie, aż do momentu przywrócenia porządku kopcowego. Nastąpi to najpóźniej z chwilą osiągnięcia liścia, tzn. węzła nie mającego synów. W programie WIE.PAS usuwanie przedziału z najmniejszym końcem jest realizowane za pomocą procedury `UsunMin`. Czas usuwania elementu minimalnego z kopca wynosi $O(\log r)$.

Rozważmy jeszcze wiersz (1) w procedurze *Ustaw*. W tym wierszu kopca Q są wstawiane przedziały o początkach w j (oznaczmy ich zbiór przez R). Aby mieć szybki dostęp do tych przedziałów porządkujemy wstępnie wszystkie przedziały ze zbioru R niemalejąco względem ich początków. Do tego celu używamy odmiany sortowania kubełkowego nazywanej **sortowaniem przez zliczanie**. W programie WIE.PAS porządkowanie to realizuje procedura `UporzadkujPrzedzialy`, działająca w czasie liniowym $O(n)$. Opis tej metody porządkowania można łatwo odczytać z opisu procedury.

Z powyższych rozważań wynika, że procedurę *Ustaw* ma rzeczywiście realizację działającą w czasie $O(n \log n)$, a co za tym idzie, oryginalny problem ustawienia wież można również rozwiązać w czasie $O(n \log n)$.

Omówienie rozwiązań podanych przez uczniów

W rozwiązaniu tego zadania uczniowie musieli pokonać dwie trudności. Po pierwsze, należało dostrzec, że zadanie można rozwiązywać niezależnie dla przedziałów poziomych i przedziałów pionowych.

Po drugie, należało zauważyć, że istnieje proste rozwiązanie problemu ustawienia dla przedziałów. Jedno podaliśmy w naszym rozwiązaniu, a inne może polegać na:

- uporządkowaniu przedziałów względem końców;
- przeglądaniu kolejnych przedziałów i wybieraniu dla każdego z nich najmniejszego reprezentanta, który nie został wcześniej wybrany; jeśli nie można tego zrobić to ustawienie nie istnieje.

Takie rozwiązanie zostało podane przez niektórych uczniów. Realizacja tej metody z użyciem kolejki priorytetowej prowadzi również do algorytmu działającego w czasie $O(n \log n)$. Osoby, które nie użyły kolejki priorytetowej, podały realizacje tych algorytmów działające w czasie $O(n^2)$.

Jeżeli, ktoś nie zauważył, że zadanie dla szachownicy można zredukować do dwóch zadań w jednym wymiarze, lub nie spostrzegł prostej metody znajdowania reprezentantów dla przedziałów, to zazwyczaj używał przeszukiwania z nawrotami, co na ogół prowadziło do algorytmów działających bardzo długo.

Niektórzy uczniowie próbowali stosować różne kryteria wyboru reprezentantów dla przedziałów, ale na ogół były to kryteria niepoprawne i zostały wychwycone podczas testowania.

Testy

Do sprawdzenia rozwiązań użyto 10 testów WIE0.IN, ..., WIE9.IN. Wśród testów były dwa, które dawały odpowiedź NIE (WIE1.IN i WIE3.IN) – testy te służyły wychwyceniu rozwiązań wykorzystujących przeszukiwanie z nawrotami.

Pozostałe testy dawały zawsze rozwiązanie pozytywne. Ich celem było sprawdzenie poprawności rozwiązań, w tym – wyeliminowanie niepoprawnych heurystyk, oraz porównanie szybkości działania poprawnych algorytmów.

Krótka charakterystyka testów:

WIE0.IN – jest przykładem z treści zadania.

WIE1.IN – test z odpowiedzią NIE dla szachownicy 100 na 100. Do dyspozycji jest tylko 50 wierszy (i kolumn), w których można postawić pierwszych 51 wież.

WIE2.IN – test z odpowiedzią TAK dla szachownicy 100 na 100. Rozwiązanie jest jednoznaczne. Kolejne wieże należy postawić na przekątnej łączącej prawy górny róg szachownicy z jej lewym dolnym rogiem.

WIE3.IN – test z odpowiedzią NIE dla szachownicy 1000 na 1000. Zadaniem tego testu było wyeliminowanie rozwiązań o wysokiej złożoności, wykorzystujących zwłaszcza przeszukiwanie z nawrotami. Liczba kolumn, w których można ustawić pierwszych 667 wież, wynosi 666.

WIE4.IN – test dużych rozmiarów z odpowiedzia TAK, z wieloma możliwościami ustawienia wież. Ten test służył eliminowaniu rozwiązań zachłannych i działających w oparciu o przeszukiwanie z nawrotami.

WIE5.IN – duży test z odpowiedzia TAK (szachownica 3000 na 3000). Dla bardzo wielu wież, pozycje, na których należało je postawić, były jednoznacznie określone, jednak ich wyznaczenie wymagało analizy możliwych ustawień pozostałych wież.

WIE6.IN – prosty test eliminujący algorytm zachłanny, który polega na ustawieniu wieży w pierwszym możliwym wierszu i w pierwszej możliwej kolumnie.

WIE7.IN – duży test z odpowiedzia TAK (szachownica 3000 na 3000) mający jednoznaczne rozwiązanie. Wymagał dokładnej analizy możliwości ustawień poszczególnych wież. Eliminował algorytmy zachłanne i z nawrotami.

WIE8.IN – prosty test eliminujący algorytmy ustawiające na pierwszym wolnym polu wieże, dla których liczba możliwych ustawień w danym momencie jest najmniejsza.

WIE9.IN – mały test (szachownica 30 na 30) z jednoznacznym rozwiązaniem dla pierwszych i ostatnich 10 wież.

7.2.3. Zadanie HAZARD (Autor: Wojciech Plandowski)

Treść zadania HAZARD

Maszyna hazardowa składa się z n generatorów liczb: $G_1, \ldots, G_n$, gdzie $1 \le n \le 1000$. Generator G_i może generować liczby tylko z ustalonego zbioru $S_i \subseteq \{1, \ldots, n\}$, lub liczbę zero oznaczającą koniec gry. Zbiór S_i może być zbiorem pustym. Niech k_i oznacza liczbę elementów zbioru S_i. Suma liczb k_i dla wszystkich $i = 1, \ldots, n$ nie może przekroczyć 12000.

Pierwsze uruchomienie generatora G_i powoduje wygenerowanie losowej liczby ze zbioru S_i. Kolejne uruchomienie powoduje wygenerowanie losowej liczby ze zbioru S_i, która nie została jeszcze wylosowana przez ten generator. Jeśli takiej liczby nie ma, G_i generuje liczbę zero.

Maszyna zaczyna działanie od uruchomienia generatora G_1. Następnie, generatory są uruchamiane zgodnie z zasadą, że jeśli uruchomiony generator wygenerował liczbę dodatnią r, to następnym uruchomionym generatorem jest G_r. Jeśli ostatnio wygenerowana liczba jest zerem, to maszyna się zatrzymuje.

Maszyna **ponosi porażkę**, gdy generator G_n wygeneruje zero oraz wszystkie generatory wygenerowały wszystkie liczby ze swoich zbiorów.

Maszyna jest **dobrze skonstruowana**, gdy istnieje kończąca się zerem sekwencja kolejno generowanych liczb nie prowadząca do porażki maszyny.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego HAZ.IN opis maszyny, tzn. liczbę generatorów n, liczby k_i i zbiory S_i,
- bada, czy maszyna opisana w pliku wejściowym jest dobrze skonstruowana,

- jeśli nie, zapisuje w pliku tekstowym HAZ.OUT jedno słowo NIE, a w przeciwnym przypadku kończącą się zerem sekwencję kolejno generowanych liczb nie prowadzącą do porażki maszyny.

Wejście

W pierwszym wierszu pliku tekstowego HAZ.IN jest zapisana jedna liczba całkowita dodatnia $n \leq 1000$. Jest to liczba wszystkich generatorów.

W $(i+1)$-szym wierszu (dla $i = 1, \ldots, n$) jest liczba k_i i po niej wszystkie elementy zbioru S_i w dowolnej kolejności. Kolejne liczby w każdym wierszu są pooddzielane pojedynczymi odstępami.

Wyjście

W pliku tekstowym HAZ.OUT należy zapisać jedno słowo NIE (jeśli maszyna nie jest dobrze skonstruowana) albo, w jednym wierszu, odpowiedni skończony ciąg liczb pooddzielanych pojedynczymi odstępami.

Przykłady

Dla pliku HAZ.IN:

```
2
2 1 2
1 2
```

poprawnym rozwiązaniem jest plik HAZ.OUT:

```
2 2 0
```

Dla pliku HAZ.IN:

```
2
1 2
0
```

poprawnym rozwiązaniem jest plik HAZ.OUT:

```
NIE
```

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

Rozwiązanie zadania HAZARD

Problem zawarty w treści zadania ma prostą interpretację w terminach grafów skierowanych. Graf skierowany jest jednoznacznie określony przez dwa

zbiory: zbiór wierzchołków i zbiór łuków, czyli par wierzchołków. W dalszych rozważaniach używamy pojęcia drogi. **Droga** w grafie jest ciągiem wierzchołków $v_0, v_1, \ldots, v_k$ takich, że (v_i, v_{i+1}) jest łukiem w grafie dla $i = 0, \ldots, k-1$. Jeśli $v_0 = v_k$, to droga nazywa się **cyklem**. Liczba k nazywana jest **długością** drogi.

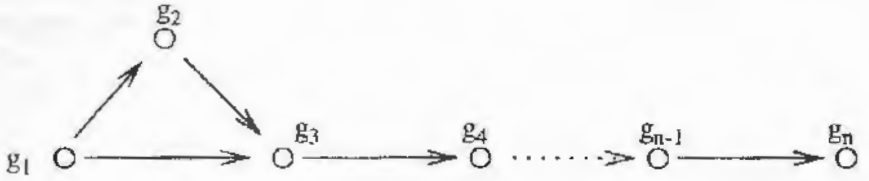
Każda maszyna hazardowa ma jednoznaczną reprezentację w postaci grafu, którego wierzchołki odpowiadają generatorom i para (g_i, g_j) tworzy łuk, jeśli $j \in S_i$, czyli jeśli g_i może spowodować uruchomienie generatora g_j, jako następnego. Tak skonstruowany graf ma n wierzchołków i $\sum_{i=1}^{n} |S_i|$ łuków, gdzie $|W|$ oznacza liczbę elementów zbioru W.

Działanie maszyny hazardowej można symulować przemieszczając się pomiędzy wierzchołkami grafu po jego łukach i drogach. Pobyt w wierzchołku g_i oznacza, że maszyna właśnie uruchomiła generator g_i. Zaczynamy więc od wierzchołka g_1. W trakcie działania maszyny, gdy generator g_i uruchamia generator g_j, to przenosimy się z wierzchołka g_i do g_j po łuku (g_i, g_j). Łuk ten istnieje, bo skoro g_i spowodował uruchomienie g_j, więc g_i musiał wygenerować liczbę j, a zatem $j \in S_i$.

W trakcie działania maszyny żaden generator nie generuje tej samej liczby dwukrotnie, a zatem, w trakcie przemieszczania się pomiędzy wierzchołkami grafu, z żadnego łuku grafu nie korzysta się dwukrotnie. Maszyna kończy swoje działanie, gdy uruchomiła generator, który wygenerował już wszystkie liczby ze swego zbioru, co w grafie odpowiada dotarciu do wierzchołka, z którego wychodzą wyłącznie wykorzystane wcześniej łuki. Maszyna przegrywa, jeśli wszystkie generatory wygenerowały wszystkie liczby ze swoich zbiorów oraz ostatnio uruchomionym generatorem był g_n. Odpowiada to w grafie przejściu wszystkich łuków oraz zakończeniu podróży w wierzchołku g_n. Podobnie, wygrana maszyny odpowiada nie wykorzystaniu jakiegoś łuku lub nie zakończeniu podróży w wierzchołku g_n.

Ciąg uruchomień kolejnych generatorów przez maszynę odpowiada drodze w grafie maszyny o początku w g_1, w której łuki się nie powtarzają i takiej, że wszystkie łuki wychodzące z ostatniego wierzchołka drogi należą do drogi. Drogę o takiej własności nazwiemy **drogą symulacyjną**. Zauważmy, że każda droga symulacyjna reprezentuje pewien możliwy ciąg uruchomień generatorów maszyny, a więc istnieje pełna odpowiedniość między drogami symulacyjnymi w grafie a ciągami generatorów uruchamianych w trakcie działania maszyny.

Działanie maszyny kończące się porażką odpowiada drodze symulacyjnej, która kończy się w g_n i która zawiera wszystkie łuki grafu. Drogi symulacyjne odpowiadające przegranej maszyny będziemy nazywać **drogami przegrywającymi**. Podobnie, drogi symulacyjne odpowiadające wygranej maszyny będziemy nazywać **drogami wygrywającymi**.



Rysunek 1. Graf nie zawierający drogi przegrywającej.

Problem zawarty w zadaniu sprowadza się do pytania czy wszystkie drogi symulacyjne w grafie maszyny hazardowej są przegrywające, czy też nie. Zauważmy, że nie wszystkie grafy zawierają drogę przegrywającą (zob. rys. 1). Okazuje się, że bardzo niewiele grafów ma takie drogi. Istnieje dość prosty warunek, który umożliwia szybkie sprawdzenie, czy graf zawiera drogę przegrywającą, czy nie.

Oznaczmy przez $we[v]$ liczbę łuków wchodzących do wierzchołak v, a przez $wy[v]$ – liczbę łuków wychodzących z v. Graf jest **spójny łukowo**, jeśli dla *każdego łuku* (u, v) *grafu istnieje droga prowadząca z* g_1 *do* u. W takim grafie każdy łuk jest osiągalny z wierzchołka g_1.

Twierdzenie 1. *Graf o* $n \geq 2$ *wierzchołkach zawiera drogę przegrywającą wtedy i tylko wtedy, gdy jest spójny łukowo i gdy spełnia następujący warunek:*

$$(\alpha)\quad \begin{cases} we[g_1] = wy[g_1] - 1, \\ we[v] = wy[v], \qquad dla\ v \notin V - \{g_1, g_n\} \\ we[g_n] = wy[g_n] + 1. \end{cases}$$

Dowód. Załóżmy, że w grafie istnieje droga przegrywająca i rozważmy wierzchołek $v \in V - \{g_1, g_n\}$. Jeśli ta droga przechodzi przez wierzchołek v dokładnie k razy, to zawiera dokładnie k łuków wchodzących do v i k łuków wychodzących z v (pamiętajmy, że łuki w drodze nie powtarzają się). Ponieważ droga przegrywająca zawiera wszystkie łuki grafu, więc $wy[v] = we[v] = k$. Podobnie dowodzimy pozostałych równości.

Załóżmy teraz, że graf jest spójny łukowo i spełnia warunek (α). Oznaczmy przez σ najdłuższą drogę symulacyjną w grafie. Przypuśćmy, że droga σ nie zawiera wszystkich łuków. Po usunięciu z grafu wszystkich łuków należących do drogi σ, graf ma przynajmniej jeden łuk i spełnia warunek

$$we[v] = wy[v] \quad \text{dla każdego } v \in V. \qquad (\beta)$$

Zauważmy, że w otrzymanym grafie istnieje wierzchołek v należący do drogi, z którego wychodzi łuk (v, u) nie należący do drogi. Gdyby taki wierzchołek nie istniał, to droga σ byłaby odizolowana od reszty grafu co przeczy warunkowi, że wszystkie łuki są osiągalne z g_1. Zaczynając od łuku (v, u) próbujemy przedłużyć drogę jak najdalej się da tak, aby łuki na niej nie powtarzały się. Ponieważ odwiedzając jakiś wierzchołek odwiedzamy jeden łuk wchodzący do niego i jeden łuk z niego wychodzącą, więc nie możemy w grafie spełniającym warunek (β) zatrzymać się w wierzchołku różnym od v. W ten sposób znajdziemy cykl przechodzący przez v i składający się z łuków nie należących do drogi σ. Rozważmy drogę, która składa się z trzech części: pierwsza prowadzi z g_1 do v tak, jak w drodze σ, druga prowadzi z v do v jak w cyklu i wreszcie trzecia część drogi prowadzi z v do g_n tak, jak w drodze σ. Droga ta jest drogą symulacyjną dłuższą od σ (o długość cyklu). Założenie, że droga σ nie zawiera wszystkich łuków doprowadziło więc do sprzeczności z założeniem, że droga σ jest najdłuższą drogą symulacyjną. Zatem σ zawiera wszystkie łuki grafu, a więc jest drogą przegrywającą. □

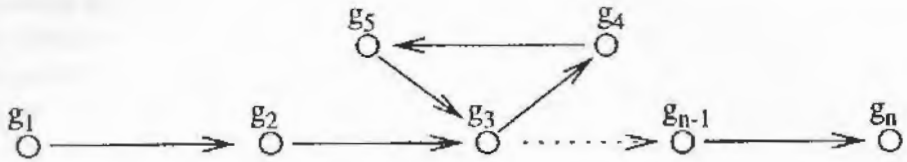
Zauważmy, że każdy graf (niekoniecznie spójny łukowo) spełniający warunek (α) ma własność, że każda droga symulacyjna kończy się w g_n. Wynika to z faktu, że przechodząc przez każdy wierzchołek wykorzystujemy jeden łuk wchodzący do wierzchołka i jeden wychodzący z wierzchołka.

Z twierdzenia 1 wynika, że w sieci niespójnej łukowo lub nie spełniającej warunkn (α) możemy budować dowolną drogę symulacyjną na ślepo wiedząc, że przegrywająca droga w tym grafie nie istnieje. Niestety, spełnienie warunku (α) nie zapewnia jeszcze nie istnienia drogi wygrywającej. Na rysumku 2 oraz 3 mamy przykłady dwóch grafów spełniających warunek (α), z których jeden zawiera, a drugi nie zawiera drogi wygrywającej.

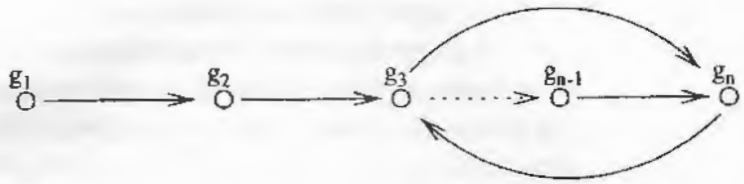
Potrzebujemy więc kryterium, które umożliwiłoby wyróżnianie wśród grafów spełniających warunek (α) tych, które nie zawierają dróg wygrywających. Intuicyjnie rzecz biorąc taki warunek nie powinien być skomplikowany, ponieważ grafy zawierające drogę przegrywającą są bardzo regularne.

Twierdzenie 2. *Niech G będzie grafem spełniającym warunek (α). W grafie G istnieje droga wygrywająca wtedy i tylko wtedy, gdy po usunięciu wierzchołka g_n pozostały graf zawiera cykl.*

Dowód. Oczywiście, jeśli w grafie istnieje cykl nie zawierający wierzchołka g_n, to istnieje w nim droga wygrywająca. Konstruujemy ją w następujący sposób. Najpierw usuwamy z grafu wyróżniony cykl i po jego usunięciu konstruujemy drogę symulacyjną w pozostałej części grafu. Zauważmy, że po usunięciu cyklu pozostały graf nadal spełnia warunek (α), a więc każda droga symulacyjna



Rysunek 2. Graf jest spójny łukowo, spełnia warunek (α) i ma jedną drogę wygrywającą $(g_1, g_2, g_3, g_6, g_7, \ldots, g_n)$ i jedną przegrywającą $(g_1, g_2, \ldots, g_n)$.



Rysunek 3. Graf jest spójny łukowo, spełnia warunek (α) i nie ma drogi wygrywającej, ale ma dwie drogi przegrywające $(g_1, g_2, \ldots, g_n, g_3, g_n)$ oraz $(g_1, g_2, g_3, g_n, g_3, g_4, \ldots, g_n)$.

kończy się w g_n. Ponieważ g_n nie należy do cyklu, więc droga ta jest także drogą symulacyjną w całym grafie. Jest to także droga wygrywająca w całym grafie, ponieważ nie zawiera żadnego łuku cyklu.

Załóżmy teraz, że istnieje wygrywająca droga w grafie spełniającym warunek (α). Zauważmy, że droga taka musi kończyć się w wierzchołku g_n, a więc ta droga zawiera wszystkie łuki wchodzące i wychodzące z g_n. Usuńmy tę drogę z grafu. W pozostałej części grafu, dla każdego wierzchołka v mamy $we[v] = wy[v]$. Oczywiście, $we[g_n] = wy[g_n] = 0$. Podobnie jak w dowodzie poprzedniego twierdzenia wnioskujemy, że w grafie tym istnieje cykl. Cykl ten oczywiście nie zawiera wierzchołka g_n. □

Na podstawie twierdzeń 1 i 2 możemy napisać teraz schemat programu HAZ.PAS, który rozwiązuje zadanie. W poniższym schemacie, procedura `WypiszNie` wypisuje do pliku wyjściowego słowo NIE, a procedura `WypiszOmijajacCykl` wypisuje ciąg liczb odpowiadający dowolnej drodze symulacyjnej, nie zawierającej łuków znalezionego wcześniej cyklu. Procedura `ZnajdzCykl` znajduje dowolny cykl nie zawierający wierzchołka g_n. Wartością funkcji logicznej `WarunekAlfa` jest `true`, jeśli graf wczytany przez procedurę `WczytajGraf` spełnia warunek (α), a `false` – w przeciwnym razie.

```
program HAZARD;
begin  {Program glowny}
```

```
 WczytajGraf;
 if n<=1 then WypiszNie
 else
  if not WarunekAlfa then
    Wypisz dowolna droge symulacyjna
  else begin
   ZnajdzCykl;
   if JestCykl then WypiszOmijajacCykl
   else WypiszNie
  end {WarunekAlfa}
end. {HAZARD}
```

Ostatnim problemem do rozwiązania jest napisanie procedury `ZnajdzCykl` – implementacja pozostałych procedur jest elementarna, zob. tekst programu HAZ.PAS. Poniżej przyjmujemy, że wierzchołek s jest **sąsiadem** wierzchołka v, jeśli $(v, s) \in E$, czyli jeśli s jest końcem łuku wychodzącego z v, a wierzchołek u jest **osiągalny** z wierzchołka v, jeśli istnieje droga prowadząca z v do u.

Algorytm wyszukujący cyklu w grafie jest modyfikacją algorytmu przeszukiwania grafu w głąb. Celem algorytmu przeszukującego graf jest odwiedzenie wszystkich wierzchołków i główną jego częścią jest procedura `Odwiedz(NrW)`, której zadaniem jest odwiedzenie wszystkich nieodwiedzonych jeszcze wierzchołków grafu, które są osiągalne z wierzchołka `NrW`. Procedura ta korzysta z faktu, że chcąc odwiedzić wierzchołki osiągalne z `NrW` wystarczy najpierw odwiedzić `NrW`, a następnie odwiedzić wszystkie wierzchołki osiągalne ze wszystkich nieodwiedzonych jeszcze jego sąsiadów (wierzchołki osiągalne z odwiedzonych sąsiadów zostały odwiedzone wcześniej).

W poniższym fragmencie algorytmu, wziętym z programu HAZ.PAS, w tablicy `Stan` jest pamiętany stan wierzchołka, którym może być: `Odwiedzony`, `NaDrodze` i `NieOdwiedzony`. Stan `Odwiedzony` oznacza, że odwiedziliśmy nie tylko ten wierzchołek, ale i wszystkie wierzchołki z niego osiągalne. Stan `NaDrodze` oznacza, że odwiedziliśmy wierzchołek, ale nie odwiedziliśmy jeszcze wszystkich wierzchołków z niego osiągalnych. Stan `NieOdwiedzony` oznacza, że nie byliśmy jeszcze w tym wierzchołku. Zauważmy, że w procedurze `ZnajdzCykl`, wierzchołki będące w stanie `NaDrodze` tworzą drogę w grafie. W poniższej fragmencie programu występuje tablica `Numer` indeksowana wierzchołkami, która numeruje wierzchołki w kolejności odwiedzenia. Tablica ta nie jest istotna w algorytmie i nie występuje w ostatecznej wersji programu HAZ.PAS, ale jest wykorzystywana w dowodzie poprawności algorytmu.

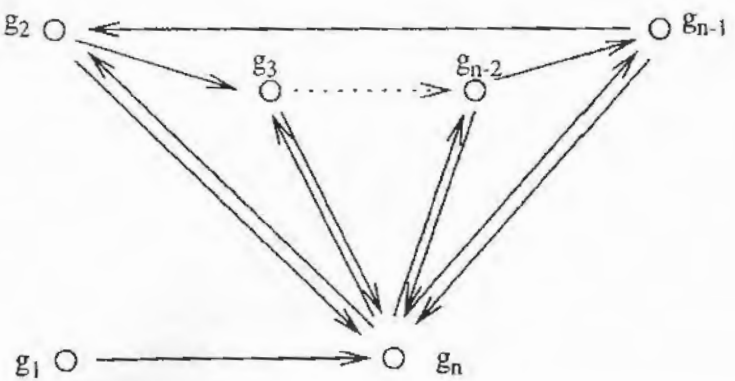
```
procedure Odwiedz(NrW:integer);
 var s:ListaSasiadow;
begin
 Stan[NrW]:=NaDrodze;
 s:=Graf[NrW];
 while (s<>nil) and (not JestCykl) do
  if s^.NrSasiada<>n then begin {luk nie prowadzi do
                                 wierzcholka gn}
   if Stan[s^.NrSasiada]=NieOdwiedzony then
    Odwiedz(s^.NrSasiada)
   else if Stan[s^.NrSasiada]=NaDrodze then begin
         JestCykl:=true;
         KoniecCyklu:=s^.NrSasiada;
         WlozDoCyklu:=true
        end; {if Stan[... }
   if not JestCykl then s:=s^.NastSasiad
  end {if s^.NrSasiada<>n }
  else {pomin luk prowadzacy do wierzcholka gn}
   s:=s^.NastSasiad;
 Stan[Nrw]:=Odwiedzony;
 Numer[Nrw]:=l; l:=l+1;
 if WlozDoCyklu then begin
  Cykl[Nrw]:=s^.NrSasiada;
  WlozDoCyklu:=Nrw<>KoniecCyklu
 end {if}
end; {Odwiedz}

procedure UstawCyklPusty;
  {Wpisuje pusty cykl do tablicy Cykl}
 var i:integer;
begin
 for i:=1 to n do Cykl[i]:= 0
end; {UstawCyklPusty}

procedure ZnajdzCykl;
  {Znajduje cykl nie przechodzacy przez wierzcholek gn}
 var i:integer;
begin
 l:=1;
```

```
 JestCykl:=false; WlozDoCyklu:=false;
 UstawCyklPusty;
 for i:=1 to n do Stan[i]:=NieOdwiedzony;
 i:=1;
 while (not JestCykl) and (i<=n-1) do begin
  if Stan[i]=NieOdwiedzony then Odwiedz(i);
  inc(i)
 end
end; {ZnajdzCykl}
```

Twierdzenie 3. *Procedura* `ZnajdzCykl` *znajduje cykl w grafie wtedy i tylko wtedy, gdy graf zawiera jakiś cykl.*

Dowód. Wystarczy dowieść, że jeśli graf zawiera cykl, to procedura `ZnajdzCykl` znajdzie jakiś cykl. Procedurę `ZnajdzCykl` można potraktować jako procedurę, która odwiedza wszystkie łuki grafu. Przy tej interpretacji sprawdzenie stanu sąsiada s wierzchołka v odpowiada odwiedzeniu łuku (v, s). Zauważmy, że każdy łuk (u, v) w grafie jest odwiedzany dokładnie jeden raz, a mianowicie w trakcie odwiedzania wierzchołka u. Jeśli w grafie istnieje cykl, to istnieje taki łuk (u, v) w cyklu, że `Numer[u]<Numer[v]`. Zastanówmy się, w którym momencie wykonywania procedury `ZnajdzCykl` odwiedzamy takie łuki. Wszystkie łuki w grafie można podzielić na trzy rozłączne podzbiory ze względu na stan ich wierzchołków końcowych w momencie odwiedzania łuków: początek jest `NaDrodze` i koniec jest `NieOdwiedzony`, początek jest `NaDrodze` i koniec jest `NaDrodze` oraz początek jest `NaDrodze` i koniec jest `Odwiedzony`. Zauważmy, że jeśli łuk (p, k) należy do pierwszej lub do trzeciej grupy, to `Numer[k]<Numer[p]`, czyli najpierw odwiedzimy wszystkie wierzchołki osiągalne z k, a dopiero potem wszystkie wierzchołki osiągalne z p. Jeśli łuk należy do grupy drugiej, to `Numer[p]<Numer[k]`, a zatem łuk (u, v) z cyklu należy do drugiej grupy łuków. Ta grupa łuków jest odwiedzana tylko w przypadku wykrycia cyklu, a więc jakiś cykl zostanie znaleziony. □

Złożoność czasowa programu HAZ.PAS jest zdominowana przez czas wykonania procedury `ZnajdzCykl`. Pozostałe procedury łatwo jest zrealizować tak, by działały w czasie $O(n+m)$, gdzie n jest liczbą generatorów maszyny i $m = \Sigma_{i=1}^{n} |S_i|$. Złożoność czasowa procedury `ZnajdzCykl` jest również $O(n+m)$, gdyż w trakcie działania algorytmu każdy łuk i każdy wierzchołek grafu jest odwiedzany dokładnie raz.



Rysunek 4. Drogami wygrywającymi w grafie są drogi symulacyjne nie zawierające żadnego łuku cyklu (2, ..., $n-1$).

Omówienie rozwiązań podanych przez uczniów

Niewielu uczniów zaprogramowało rozwiązanie podobne do powyższego. W większości rozwiązań przeglądano wszystkie możliwe drogi symulacyjne. Bez względu na sposób implementacji, metoda ta prowadzi do algorytmów mało efektywnych w przypadku grafów, w których liczba dróg wygrywających jest niewielka w stosunku do liczby dróg przegrywających. Graf na rys. 4 zawiera $(n-2)!$ dróg wygrywających (te z dróg symulacyjnych, które pomijają łuki cyklu $(2, 3, \ldots, n-1)$ oraz $(n-2)!\,(2^{n-2}-1)$ dróg przegrywających, a zatem na jedną drogę wygrywającą przypada $(2^{n-2}-1)$ dróg przegrywających.

Duża liczba uczniów użyła niepoprawnego algorytmu. Programy z tej grupy budowały jedną drogę symulacyjną w nadziei, że na jej podstawie da się zbudować drogę wygrywającą. Drogę budowano albo losowo wybierając jej kolejne łuki, albo używając rozmaitych heurystyk. Jeśli znaleziona droga zawierała cykl nie przechodzący przez wierzchołek g_n, to ten cykl był usuwany i powstała droga była wygrywająca. W przeciwnym przypadku sprawdzano, czy skonstruowana droga jest wygrywająca. Jeśli droga nie była wygrywająca, to uznawano, że maszyna jest źle skonstruowana. Graf na rys. 4 jest przykadem, dla którego programy z tej grupy mogą dawać złe wyniki. Graf ten zawiera drogi przegrywające, które nie zawierają cyklu przechodzącego przez wierzchołek g_n. Jeśli program znajduje jedną z takich dróg, to stwierdza, że maszyna jest źle skoustruowana. Można obliczyć, że liczba dróg symulacyjnych, dla których te programy znajdują poprawny wynik, jest wykładniczo mała w stosunku do wszystkich dróg symulacyjnych.

Testy

Programy uczniów były sprawdzane na dwunastu testach. Dwa testy HAZ0.IN i HAZ1.IN pochodziły z treści zadania. Pozostałe testy można podzielić na trzy grupy: testy badające przypadki szczególne, testy poprawnościowe oraz testy złożonościowe. Dalej, testy opisujemy jako grafy maszyny.

Pośród testów badających przypadki szczególne znalazły się następujące grafy:

HAZ2.IN – graf jednowierzchołkowy z jednym łukiem (minimalna możliwa liczba wierzchołków grafu); w tym przypadku maszyna jest źle skonstruowana;

HAZ3.IN – graf sześciowierzchołkowy, nie zawierający żadnego łuku; w tym przypadku maszyna jest dobrze skonstruowana i zawiera jedną drogę wygrywającą: g_1 (najkrótsza z możliwych dróg wygrywających);

HAZ4.IN – graf tysiącwierzchołkowy (maksymalna możliwa liczba wierzchołków), z cyklem przed ostatnim wierzchokiem, mający rozwiązanie.

Druga grupa testów, testy poprawnościowe, obejmowała m.in. graf HAZ5.IN, w którym po usunięciu wierzchołka g_n część wierzchołków staje się nieosiągalna z wierzchołka g_1 oraz graf HAZ6.IN, w którym po usunięciu wierzchołka g_n pozostaje tylko jeden cykl.

Zadaniem testów złożonościowych było wychwycenie algorytmów o złożoności liniowej, kwadratowej i więcej niż kwadratowej – dwa z nich miały po ok. 1000 wierzchołków i ok. 12000 łuków. Pośród tych testów były zarówno takie, które odpowiadały maszynom dobrze skonstruowanym jak i takie, które odpowiadały maszynom źle skonstruowanym. W grafach odpowiadających maszynom dobrze skonstruowanym, liczba dróg wygrywających była minimalna w stosunku do liczby dróg przegrywających.

7.2.4. Zadanie SŁOWA FIBONACCIEGO (Autor: Wojciech Rytter)

Treść zadania SŁOWA FIBONACCIEGO

Słowa Fibonacciego definiujemy podobnie jak liczby Fibonacciego

$$FIB_1 = \text{b};\ FIB_2 = \text{a};\ FIB_{k+2} = FIB_{k+1} \oplus FIB_k \text{ dla } k \geq 1,$$

gdzie $\oplus$ jest znakiem operacji łączenia (konkatenacji) słów.

Mamy zatem

$$FIB_3 = \text{ab};\ FIB_4 = \text{aba};\ FIB_5 = \text{abaab};\ FIB_6 = \text{abaababa}.$$

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego FIB.IN słowo złożone z co najmniej jednej i co najwyżej 30 liter a lub b oraz liczbę całkowitą dodatnią $n \leq 200$,
- oblicza, ile razy dane słowo występuje w n-tym słowie Fibonacciego FIB_n (poszczególne wystąpienia danego słowa mogą częściowo na siebie zachodzić),
- zapisuje wynik w pliku tekstowym FIB.OUT.

Wejście

W pierwszym wierszu pliku tekstowego FIB.IN jest zapisane jedno słowo złożone z co najmniej jednej i co najwyżej 30 małych liter a lub b.

W drugim wierszu jest zapisana jedna liczba całkowita dodatnia $n \leq 200$.

Wyjście

W pliku tekstowym FIB.OUT należy zapisać jedną liczbę całkowitą nieujemną, która jest liczbą wystąpień danego słowa w n-tym słowie Fibonacciego FIB_n.

Przykład

Dla pliku FIB.IN:

aba
6

w pliku FIB.OUT należy zapisać jedną liczbę:

3

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

Rozwiązanie zadania SŁOWA FIBONACCIEGO

Wczytane słowo oznaczmy przez z i nazwijmy **wzorcem**. Niech l będzie długością wzorca, a więc $l \leq 30$. Istotną trudnością w zadaniu jest potencjalnie astronomiczny rozmiar rozważanych słów Fibonacciego FIB_n, przy jednocześnie niewielkim rozmiarze wzorca z. Nie możemy zatem wyszukiwać z w FIB_n w sposób bezpośredni, natomiast możemy to łatwo robić w istotnych fragmentach słowa FIB_n, o rozmiarze proporcjonalnym do długości z.

W opisie rozwiązania tego zadania oraz w programie FIB.PAS korzystamy z następujących funkcji języka Turbo-Pascal:

- `pos(x,y)` jest pierwszą (od lewej strony) pozycją w słowie `y`, od której zaczyna się wystąpienie słowa `x`.
- `delete(x,i,j)` jest operacją usuwającą ze słowa `x` fragment zaczynający się na pozycji `i` i kończący się na pozycji `j`.
- `concat(x,y)` jest złożeniem słów `x` i `y`.
- `copy(x,i,j)` jest fragmentem słowa `x` zaczynającym się na pozycji `i` i kończący się na pozycji `j`.
- `length(x)` jest długością słowa `x`.

Podstawowa funkcja w programie ma nazwę `LiczbaWyst` i wyznacza liczbę wystąpień słowa `Wzorzec` w danym słowie `y`. Wartość tej funkcji może być obliczana iteracyjnie:

```
function LiczbaWyst(y:string):longint;
 var Licznik,p:longint;
begin
 Licznik:=0;
 while pos(Wzorzec,y)>0 do begin
  inc(Licznik);
  delete(y,1,pos(Wzorzec,y))
 end;
 LiczbaWyst:=Licznik
end; {LiczbaWyst}
```

Można również zrealizować tę funkcję rekurencyjnie w następujący sposób:

```
function LiczbaWyst(y:string):longint;
begin
 if pos(wzorzec,y)=0 then LiczbaWyst:=0
 else LiczbaWyst:=1+LiczbaWyst(delete(y,1,pos(wzorzec,y)))
end; {LiczbaWyst}
```

Dla wzorca o dużej długości są znane znacznie efektywniejsze algorytmy wyznaczania wszystkich jego wystąpień w innym słowie. Jednym z nich jest algorytm podany przez Knutha, Morrisa i Pratta (algorytm KMP). Ponieważ jednak w naszym przypadku wzorzec jest dość krótki, poprzestaniemy na podanych dwóch wersjach funkcji `LiczbaWyst`.

Rozwiązaniem zadania jest wartość funkcji `LiczbaWyst` wyznaczona dla argumentu FIB_n, jednakże bezpośrednie użycie tej funkcji na ogół nie jest moż-

liwe ze względu na bardzo duże długości słów FIB_n. Funkcję tę używamy jedynie pomocniczo dla małych n lub dla krótkich fragmentów słów FIB_n.

Ze względu na możliwość wystąpienia w obliczeniach dużych liczb, w programie została zdefiniowana arytmetyka, w której liczby, tzw. **długie liczby**, mogą mieć co najwyżej `LiczbaCyfr` cyfr w postaci dziesiętnej. Długa liczba jest pamiętana jako tablica swoich cyfr. Przyjęto na podstawie warunków zadania, że stała `LiczbaCyfr` ma wartość 50. W obliczeniach są wykorzystywane następujące procedury, działające w tej arytmetyce:

- `Przypisz(y,x)` – liczba `x` typu `interger` zostaje zamieniona na długą liczbę `y`;
- `WypiszWynik(l)` – wyprowadza wartość długiej liczby `l` cyfra po cyfrze;
- `Dodaj(a,b,c)` – długa liczba `c` jest sumą długich liczb `a` i `b`.

Algorytm zrealizowany w programie najpierw znajduje najmniejsze m takie, że długość słowa Fibonacciego FIB_m nie jest mniejsza od l, gdzie l jest długością wzorca. Odpowiedni fragment programu jest następujący:

```
m:=1; x:='b'; y:='a';
while length(x) < l do begin
 inc(m); z:=y; y:=concat(y,x); x:=z
end;
```

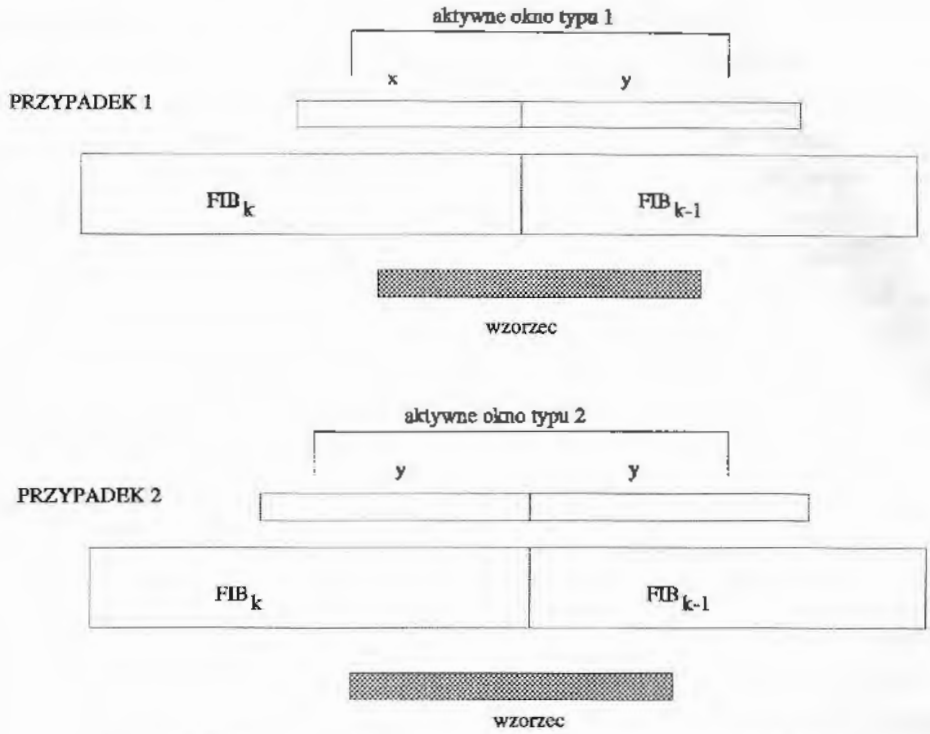
Niech $k > m$ i rozważmy wszystkie takie możliwe położenia wzorca w słowie FIB_{k+1}, że część wzorca jest w FIB_k, a część jest w FIB_{k-1}, zob. rys. 1. Wzorzec mieści się wtedy w **oknie**, które jest złożeniem sufiksu słowa FIB_k i prefiksu słowa FIB_{k-1}, oba o długości $l-1$. Zauważmy, że ze sposobu budowania kolejnych słów Fibonacciego wynika, iż są możliwe jedynie dwa okna, jedno złożone z sufiksu słowa $x = FIB_m$ i prefiksu słowa $y = FIB_{m+1}$, a drugie złożone z prefiksu słowa y dopisanego do siebie, zob. rys. 1. Oznaczmy słowa odpowiadające tym oknom przez `Okno1` i `Okno2`. Słowa te są wyznaczane w intrukcjach:

```
Okno1:=concat(copy(x,length(x)-l+2,l-1),copy(y,1,l-1));
Okno2:=concat(copy(y,length(y)-l+2,l-1),copy(y,1,l-1));
```

Następnie, liczbę wystąpień wzorca w oknach przypisujemy zmiennym `q1` oraz `q2` w instrukcjach:

```
Przypisz(q1,LiczbaWyst(Okno1));
Przypisz(q2,LiczbaWyst(Okno2));
```

Zasadnicza część programu ma następującą postać:



Rysunek 1. Dwa możliwe okna i położenia wzorca wewnątrz okna.

```
Przelacznik:=false;
while m < n do begin
 Przelacznik:=not Przelacznik;
 inc(m); Pom:=j; Dodaj(i,j,j); i:=Pom;
 if Przelacznik then Dodaj(j,q2,j) else Dodaj(j,q1,j)
end;
```

Przed każdym wejściem do pętli `while` prawdziwy jest niezmiennik:

`i=LiczbaWyst(`FIB_m`)`, `j=LiczbaWyst(`FIB_{m+1}`)`.

Wartości liczb i oraz j są wyznaczane ze wzoru:

`LiczbaWyst(`FIB_{m+1}`)=LiczbaWyst(`FIB_m`)+LiczbaWyst(`FIB_{m-1}`)+q`,

gdzie `q` jest równe naprzemiennie `q1` lub `q2`.

Na końcu program wypisuje wynik `i=LiczbaWyst(`FIB_n`)`.

Omówienie rozwiązań podanych przez uczniów

Uczniowie zaprojektowali algorytmy iteracyjne lub rekurencyjne. Istotne dla oceny rozwiązania było to, czy algorytm pamiętał poprzednio obliczone wartości – wtedy działał bardzo szybko. Jeśli natomiast wielokrotnie wyznaczał liczbę wystąpień w tym samym słowie Fibonacciego, to działał dość długo.

Nie wszyscy uczniowie zauważyli, że są tylko dwa możliwe okna i wyznaczali liczbę wystąpień wzorca na styku kolejnych słów Fibonacciego za każdym razem. Spowalniało to znacznie działanie programu w takich przypadkach.

Niektórzy posłużyli się algorytmem Knutha-Morrisa-Pratta do obliczania wartości funkcji `LiczbaWyst`.

Testy

Jak zwykle, użyto dwóch grup testów – sprawdzających poprawność zrealizowanych algorytmów oraz ich efektywność.

Ponadto, ponieważ liczba wystąpień wzorca może być bardzo duża, celem grupy testów było sprawdzenie, czy w programie została zrealizowana własna arytmetyka.

Test FIB0.IN jest przykładem z treści zadania, a dwa następne testy badały poprawność programów dla brzegowych danych, gdy $n = 1$. FIB3.IN jest prostym testem poprawnościowym, a w FIB4.IN, wzorzec jest dłuższy niż słowo F_6. W teście FIB5.IN wzorzec składa się z jednej litery a. Test FIB6.IN zawiera długi wzorzec występujacy w słowie F_{30} prawie 30000 razy. Z kolei, wzorzec w teście FIB7.IN, abaabba, nie występuje w ogóle w słowie F_{100}.

W trzech ostatnich testach rozwiązaniami są liczby o ponad 40 cyfrach, a wśród nich, w teście FIB10.IN, wzorzec a występuje w słowie F_{200} największą możliwą liczbę razy.

7.3. Zawody III stopnia

7.3.1. Zadanie PERMUTACJE ANTYARYTMETYCZNE (Autor: Wojciech Guzicki)

Treść zadania PERMUTACJE ANTYARYTMETYCZNE

Permutację $p_0, p_1, \ldots, p_{n-1}$ liczb naturalnych $0, 1, \ldots, n-1$ nazywamy permutacją antyarytmetyczną, jeśli nie występuje w niej żaden trzywyrazowy ciąg arytmetyczny, tzn. nie istnieją takie trzy indeksy $i < j < k$, że liczby p_i, p_j, p_k (w tej kolejności) tworzą ciąg arytmetyczny. Na przykład, ciąg liczb 3, 1, 0, 4, 2 jest permutacją antyarytmetyczną liczb 0, 1, 2, 3, 4. Natomiast ciąg 0, 5, 4, 3, 1, 2 nie jest permutacją antyarytmetyczną, ponieważ jego wyrazy – pierwszy, pią-

ty i szósty: 0, 1, 2 tworzą ciąg arytmetyczny (także wyrazy drugi, czwarty i piąty: 5, 3, 1 oraz drugi, trzeci i czwarty: 5, 4, 3 tworzą ciągi arytmetyczne).

Zadanie

Ułóż program, który:

- wczytuje z pliku tekstowego PER.IN jedną liczbę naturalną n,
- znajduje i zapisuje w pliku tekstowym PER.OUT jedną dowolną permutację antyarytmetyczną liczb 0, 1, ..., $n - 1$.

Wejście

W pliku tekstowym PER.IN jest zapisana jedna liczba całkowita n spełniająca nierówność $3 \leq n \leq 1000000$.

Wyjście

W n kolejnych wierszach pliku tekstowego PER.OUT należy zapisać n różnych liczb całkowitych ze zbioru $\{0, 1, ..., n - 1\}$, każdą w osobnym wierszu. Liczby w kolejnych wierszach powinny tworzyć permutację antyarytmetyczną liczb 0, 1, ..., $n - 1$.

Przykład

Jeśli w pliku PER.IN jest zapisana liczba 5, to poprawnym rozwiązaniem jest następujący plik PER.OUT:

```
3
1
0
4
2
```

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

Rozwiązanie zadania PERMUTACJE ANTYARYTMETYCZNE

Najprościej jest napisać rozwiązanie rekurencyjne. Pokażemy najpierw metodą indukcji matematycznej, że dla każdej liczby naturalnej l istnieje permutacja antyarytmetyczna liczb 0, 1, 2, ..., $2^l - 1$. Sposób konstrukcji ciągu wyniknie z dowodu.

Dla $l = 0$ bierzemy permutację 0. Oczywiście jest to permutacja antyarytmetyczna.

Załóżmy, że mamy już permutację antyarytmetyczną

$$p_0, p_1, \ldots, p_m$$

liczb 0, 1, ..., m, gdzie $m = 2^l - 1$. Tworzymy wtedy permutację

$$2p_0, 2p_1, \ldots, 2p_m, 2p_0 + 1, 2p_1 + 1, \ldots, 2p_m + 1. \tag{1}$$

Pokażemy teraz, że jest to permutacja antyarytmetyczna liczb 0, 1, ..., $2m + 1$. Zauważmy przy tym, że $2m + 1 = 2^{l+1} - 1$.

Sprawdzenie, że otrzymaliśmy rzeczywiście permutację liczb 0, 1, ..., $2m + 1$, jest bardzo łatwe i pominiemy je. Przypuśćmy, że $i < j < k$ oraz p_i, p_j, p_k tworzą ciąg arytmetyczny. Wtedy liczby p_i i p_k są jednakowej parzystości (bo $p_k = a_i + 2r$, gdzie r jest różnicą tego ciągu arytmetycznego). Ponieważ liczba p_j występuje w tej permutacji między liczbami p_i i p_k, więc jest tej samej parzystości co te dwie liczby. Możliwe są więc dwa przypadki:

Przypadek 1. Liczby p_i, p_j, p_k są parzyste. Wtedy $p_i = 2p_s$, $p_j = 2p_t$, $p_k = 2p_u$ dla pewnych $s, t, u \le m$. Liczby p_s, p_t i p_u tworzą ciąg arytmetyczny, co przeczy założeniu o tym, że permutacja $p_0, p_1, \ldots, p_m$ jest permutacją antyarytmetyczną.

Przypadek 2. Liczby p_i, p_j, p_k są nieparzyste. Wtedy $p_i = 2p_s + 1$, $p_j = 2p_t + 1$, $p_k = 2p_u + 1$ dla pewnych $s, t, u \le m$. Liczby p_s, p_t i p_u również tworzą ciąg arytmetyczny, co znów przeczy założeniu o permutacji $p_0, p_1, \ldots, p_m$.

Sposób konstrukcji permutacji antyarytmetycznych jest już widoczny. Zaczynamy od permutacji jednoelementowej 0. Następnie, powtarzając krok indukcyjny, tworzymy następne permutacje:

$$2\cdot0 = 0,\ 2\cdot0+1 = 1,$$
$$2\cdot0 = 0,\ 2\cdot1 = 2,\ 2\cdot0+1 = 1,\ 2\cdot1+1 = 3,$$
$$2\cdot0 = 0,\ 2\cdot2 = 4,\ 2\cdot1 = 2,\ 2\cdot3 = 6,\ 2\cdot0+1 = 1,\ 2\cdot2+1 = 5,\ 2\cdot1+1 = 3,\ 2\cdot3+1 = 7,$$

czyli

$$0,\ 1,$$
$$0,\ 2,\ 1,\ 3,$$
$$0,\ 4,\ 2,\ 6,\ 1,\ 5,\ 3,\ 7.$$

Oczywiście, jeśli mamy utworzyć permutację antyarytmetyczną liczb 0, 1, ..., $n - 1$, gdzie $n < 2^l$, to wystarczy pewne liczby otrzymanej permutacji pominąć (mianowicie liczby $p_i \ge n$).

Bezpośrednia realizacja powyższego rozwiązania wymaga jednak pamiętania kolejnych ciągów. W przypadku, gdy dysponujemy tylko ograniczoną pamięcią, a tak jest w naszym zadaniu (gdyż $0 \le n \le 1\,000\,000$), musimy spróbować

generować kolejne liczby bez zapamiętywania poprzednich. W tym celu przyjrzyjmy się dokładniej sposobowi konstrukcji kolejnych permutacji.

Z permutacji $p_0, p_1, p_2, \ldots, p_m$ tworzyliśmy permutację (1).

Pomnożenie liczby p przez 2 oznacza w dwójkowym systemie pozycyjnym dopisanie zera na końcu. Utworzenie z liczby p liczby $2p + 1$ oznacza dopisanie na końcu jedynki. Popatrzymy więc na rozwinięcia dwójkowe liczb $p_0, p_1, p_2, \ldots, p_m$. Niech $m = 2^l - 1$. Rozwinięcia te mają zatem po l bitów. Nowy ciąg tworzymy dopisując na końcu każdej z naszych liczb zero, a następnie powtarzając ten sam ciąg, ale tym razem dopisując na końcu jedynkę. Zatem w ciągu o postaci (1) będą na początku stały liczby z zerem na końcu, a po nich liczby z jedynką na końcu. Pamiętamy teraz, że ciąg $p_0, p_1, p_2, \ldots, p_m$ został utworzony w taki sam sposób. A więc również w nim na początku stoją liczby mające ostatnią cyfrę równą 0, a po nich stoją liczby mające ostatnią cyfrę równą 1. W nowym ciągu mamy więc następującą kolejność liczb: najpierw kończące się cyframi 00, potem kończące się cyframi 01, potem cyframi 10 i wreszcie kończące się cyframi 11. I tak dalej.

Widzimy więc, że wszystkie liczby mające l cyfr w rozwinięciu dwójkowym są ustawione w następującej kolejności. Dla $l = 0$:

0

dla $l = 1$:

0
1

dla $l = 2$:

00
10
01
11

i ogólnie:

0000...00
1000...00
0100...00
1100...00
0010...00
1010...00
...
0111...11
1111...11

i tak dalej. Jest to dokładnie ta sama kolejność, w jakiej występują liczby naturalne 0, 1, 2, ..., z tym tylko, że cyfry w rozwinięciu dwójkowym tych kolej-

nych liczb zostały „odwrócone": pierwsza stała się ostatnią, druga przedostatnią i tak dalej.

Teraz napisanie algorytmu jest już łatwe. Ustalamy liczbę l, będącą długością rozwinięcia dwójkowego naszych liczb. Badamy kolejno liczby od 0 do $2^l - 1$. Każdą z tych liczb rozwijamy w układzie dwójkowym, rozwinięcie odwracamy i otrzymujemy kolejną liczbę naszej permutacji antyarytmetycznej; należy przy tym pamiętać, że jeśli otrzymana liczba jest większa lub równa n, to ją pomijamy.

Ten algorytm został zrealizowany w programie PER.PAS w postaci następujacych kroków:

1. Wczytaj liczbę n z pliku tekstowego PER.IN.
2. Ustal długość l rozwinięcia dwójkowego liczby n. W funkcji `Dlugosc` został wykorzystany „szkolny" algorytm przedstawiania liczby w systemach pozycyjnych: dzielimy n przez 2, iloraz dzielimy przez 2 itd. Nie trzeba pamiętać reszt, gdyż potrzebna jest tylko liczba dzieleń równa długości rozwinięcia.
3. W pętli, kolejne liczby od 0 do $2^l - 1$ przedstaw w systemie dwójkowym, z tym tylko, że otrzymywane cyfry zapisuj na początku rozwinięcia, a nie na końcu. Otrzymane rozwinięcia „zwij" stosując algorytm Hornera. Jeśli wynik jest mniejszy od n, to zapisz go w pliku PER.OUT.

Pełną realizacją tego algorytmu jest program PER.PAS.

Uwagi końcowe

Problem istnienia permutacji antyarytmetycznych ma interesujące uogólnienia nieskończone. Ciąg nieskończony $p_0, p_1, p_2, \ldots, p_m, \ldots$ nazwiemy ciągiem n-antyarytmetycznym, jeśli jego wyrazy są liczbami naturalnymi (włącznie z zerem), każda liczba naturalna występuje w tym ciągu dokładnie jeden raz oraz nie istnieją indeksy $i_1 < i_2 < \ldots < i_n$ takie, że liczby $p_{i_1}, p_{i_2}, \ldots, p_{i_n}$ tworzą ciąg arytmetyczny. Można łatwo dowieść, że nie istnieją nieskończone ciągi 3-antyarytmetyczne. Nieco trudniej dowieść (por. J.A. Davis, R.C. Entringer, R.L. Graham, G.J. Simmons, On Permutations Containing No Long Arithmetic Prograssions, *Acta Arithmetica* 34(1977), 81–90), że istnieje ciąg 5-antyarytmetyczny. Problem istnienia ciągów 4-antyarytmetycznych wydaje się być nadal otwarty.

Omówienie rozwiązań podanych przez uczniów

Zadanie rozwiązało poprawnie kilkunastu uczniów. Czterech z nich podało rozwiązanie omówione powyżej. Dziesięciu podało różne rozwiązania rekurencyjne. Sprowadzały się one do następującego rozumowania.

Zamiast permutacji liczb $1, 2, \ldots, n-1$ rozważmy permutację liczb $p, p+r, p+2r, \ldots, p+(n-1)r$ tzn. permutację wyrazów ciągu arytmetycznego o pierwszym wyrazie p i różnicy r. Zauważamy, podobnie jak powyżej, że jeśli liczby $p+ir, p+jr, p+kr$ tworzą ciąg arytmetyczny, to i i k są jednakowej parzystości. Zatem wystarczy wypisać wyrazy naszego ciągu arytmetycznego w następującej kolejności: najpierw wyrazy postaci $p+ir$ dla i parzystych, potem dla i nieparzystych. W ramach każdej grupy zadbamy o to, by tworzyły one permutację antyarytmetyczną. Ten pomysł można zrealizować za pomocą następującej procedury rekurencyjnej:

```
var We,Wy:text; {Pliki: wejsciowy i wyjsciowy}

procedure Wypisuj(p,r,n:longint);
 {Procedura wypisuje permutacje antyarytmetyczne liczb
  o postaci p+i*r dla 0 <= i <= n-1.}
begin
 if n=1 then writeln(Wy,p)
 else begin
  {Rozpatrujemy liczby o postaci p+i*r dla 0 <= i <= n-1}
  Wypisuj(p,2*r,(n+1) div 2);
   {jest (n+1) div 2 liczb w postaci p+i*r z parzystym i}
  Wypisuj(p+r,2*r,n div 2)
  {jest n div 2 liczb w postaci p+i*r z nieparzystym i}
 end {else}
end; {Wypisuj}
```

Procedurę tę należy wywołać z parametrami `Wypisuj(0,1,n)`.

Niektórzy zawodnicy powtarzali rozwiązanie naszkicowane w dowodzie, zdając sobie sprawę z trudności wynikających z braku pamięci. Wyniki pośrednie próbowali zatem zapisywać na dysku, co powodowało, że ich programy działały bardzo długo.

Wśród rozwiązań złych znalazły się następujace rozwiązania. Zawodnicy znaleźli kilka rozwiązań dla małych wartości n, stablicowali je i program wypisywał je dla tych n. Dla n większych program po prostu nic nie robił. Oczywiście taki program nie może być uznany za pełne rozwiązanie zadania, jednak otrzymywał kilka punktów, gdyż poprawnie przechodził przez niektóre testy dla małych n.

Testy

Dane dla programu PER.PAS stanowi jedna liczba, n. Dla $n = 16$ w pliku PER.OUT program umieszcza permutację: 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15, która ma postać (1) i jest permutacją antyarytmetyczną. Dla $n = 10$ rozwiązaniem jest permutacja: 0, 8, 4, 2, 6, 1, 9, 5, 3, 7, która powstaje z permutacji dla $n = 16$ przez usunięcie z niej liczb równych co najmniej 9, a zatem jest to również permutacja antyarytmentyczna.

7.3.2. Zadanie AGENCI (Autor: Marcin Kubica)

Treść zadania AGENCI

W kraju działają agenci obcych wywiadów. Zajmują się oni nie tylko wykradaniem tajemnic, ale także nawzajem się szpiegują. Mówimy, że agent A rozpracował agenta B, jeśli A posiada dokumenty wystarczające do aresztowania B.

Niektórzy agenci są przekupni – w zamian za odpowiednią kwotę pieniędzy są gotowi oddać wszystkie posiadane dokumenty. Aresztując agenta przechwytujemy wszystkie zgromadzone przez niego dokumenty. Przekupienie odpowiednio wybranych agentów może więc uruchomić łańcuch aresztowań prowadzący do zlikwidowania wszystkich agentów działających w kraju.

Rodzimy kontrwywiad dostarczył nam informacje o tym, jacy obcy agenci działają w kraju, którzy z nich są przekupni i za jaką cenę, a także, którzy agenci rozpracowali których.

Liczba agentów $n \leq 3000$; są oni ponumerowani od 1 do n.

Zadanie

Ułóż program, który:

- wczytuje z pliku tekstowego AGE.IN dane zgromadzone przez kontrwywiad,
- stwierdza, czy jest możliwe, by przez przekupienie odpowiednio wybranych agentów uzyskać informacje, uruchamiające łańcuch prowadzący do aresztowania wszystkich agentów w kraju i jeśli tak, oblicza minimalny koszt takiego zakupu, a jeśli nie, znajduje numer jednego z agentów, którego nie będzie można aresztować,
- zapisuje wynik do pliku tekstowego AGE.OUT.

Wejście

W pierwszym wierszu pliku tekstowego AGE.IN jest zapisana jedna liczba naturalna n. Jest to liczba wszystkich działających w kraju agentów, $1 n \leq 3000$.

W drugim wierszu jest zapisana jedna liczba naturalna p. Jest to liczba wszystkich agentów przekupnych, $1 \le p \le n$.

W kolejnych p wierszach są zapisane informacje o przekupnych agentach. W każdym takim wierszu są zapisane dwie liczby całkowite dodatnie oddzielone odstępem – pierwsza z nich jest numerem przekupnego agenta, a druga to kwota, za którą można go przekupić – jest to liczba nie większa niż 20000.

Kolejny wiersz zawiera jedną liczbę r, gdzie $1 \le r \le 8000$. Jest to liczba takich par (A,B), że agent A rozpracował agenta B.

W następnych r wierszach podane są wszystkie takie pary. W każdym z tych wierszy są zapisane dwie różne liczby ze zbioru $\{1, 2, \ldots, n\}$ oddzielone odstępem – pierwsza, to numer agenta, który rozpracował, a druga, to numer agenta, który został rozpracowany.

Wyjście

W pierwszym wierszu pliku tekstowego AGE.OUT należy zapisać jedno słowo: TAK – jeśli istnieje możliwość aresztowania wszystkich agentów działających w kraju, albo NIE – jeśli taka możliwość nie istnieje.

W drugim wierszu należy zapisać jedną liczbę całkowitą dodatnią: minimalny koszt zakupu informacji wystarczających, by doprowadzić do aresztowania wszystkich agentów w kraju, albo numer dowolnego agenta, którego nie da się zaaresztować.

Przykłady

Dla pliku AGE.IN:

```
3
2
1 10
2 100
2
1 3
2 3
```

w pliku AGE.OUT należy zapisać:

```
TAK
110
```

Dla pliku AGE.IN:

```
4
2
1 100
4 200
```

2
1 2
3 4

w pliku AGE.OUT należy zapisać:

NIE
3

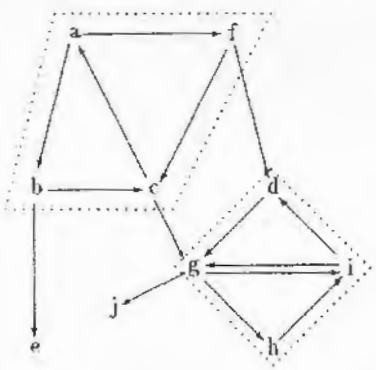
Twój program powinien szukać pliku AGE.IN w katalogu bieżącym i tworzyć plik AGE.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę AGE.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania.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

Rozwiązanie zadania AGENCI

Analiza problemu

Kluczem do rozwiązania zadania jest zrozumienie zależności między agentami, które wynikają z tego, że jedni agenci posiadają dokumenty mogące doprowadzić do aresztowania innych agentów. Zależności te możemy przedstawić w postaci **grafu skierowanego**: prowadzimy **łuk** $x \to y$ (taki łuk oznaczamy również przez (x, y)), gdy agent x ma dokumenty dotyczące agenta y. Przykładowy graf jest pokazany na rys. 1.

Każdy graf skierowany można podzielić na części (**podgrafy**), w obrębie których istnieje przejście między dwoma dowolnymi wierzchołkami tam i z powrotem. (W terminach teorii grafów, przejście między dwoma wierzchołkami wykorzystujące istniejące łuki nazywa się **drogą**). Części takie nazywamy **silnie spójnymi składowymi** grafu lub składowymi silnej spójności. W grafie na



Rysunek 1. Graf skierowany z podziałem na silnie spójne składowe.

rys. 1 mamy cztery silnie spójne składowe: $\{a, b, c, f\}$, $\{d, g, h, i\}$ oraz dwie składowe jednowierzchołkowe $\{e\}$ i $\{j\}$. Każdy graf można zawsze podzielić na takie składowe, gdyż relacja między wierzchołkami x i y: „istnieje droga z wierzchołka x do y oraz droga z y do x" jest relacją równoważności. Silnie spójne składowe są klasami abstrakcji tej relacji. Zauważmy, że w każdej składowej wystarczy aresztować (lub przekupić) jednego agenta, aby wszyscy pozostali zostali aresztowani.

Przyjrzyjmy się teraz łukom łączącym różne składowe A i B. Oznaczmy przez $A \propto B$ sytuację, kiedy możemy przejść od pewnego agenta $a \in A$ do pewnego agenta $b \in B$, a więc, gdy w grafie istnieje droga z a do b. Zauważmy, że wystarczy wówczas aresztować (lub przekupić) dowolnego agenta w składowej A, aby wszyscy agenci w składowej B zostali aresztowani. Relacja $\propto$ jest relacją **częściowego porządku** określoną na silnie spójnych składowych, gdyż jest **antysymetryczna** i **przechodnia**. Wynika to stąd, że gdybyśmy mieli $A \propto B$ oraz $B \propto A$, to można by przejść od składowej A do składowej B oraz od składowej B do A, czyli agenci z A i B byliby w jednej silnie spójnej składowej. Ponadto, jeśli $A \propto B$ oraz $B \propto C$, to $A \propto C$. Wyróżnijmy pewien rodzaj składowych. Silnie spójna składowa A w grafie jest **minimalna**, jeśli nie istnieje żadna inna składowa A' ją poprzedzająca, czyli spełniająca $A \neq A'$ i $A' \propto A$. Zauważmy, że wystarczy przekupić po jednym agencie w każdej mimimalnej składowej grafu. Ponadto jedyną metodą aresztowania agentów z takiej składowej jest przekupienie jednego z nich. Zatem w każdej minimalnej składowej należy przekupić najtańszego agenta. Jeśli jednak w którejś minimalnej składowej wszyscy agenci są nieprzekupni, to nie da się ich aresztować i odpowiedź w rozwiązaniu zadania powinna brzmieć NIE. Jako numer agenta, którego nie da się aresztować, wystarczy podać numer dowolnego agenta z takiej składowej.

Rozwiązanie optymalne

Opiszemy teraz bardziej szczegółowo rozwiązanie oparte na wynikach powyższych rozważań. Algorytm ma następującą postać:

1. Zbuduj skierowany graf zależności między agentami.
2. Wyznacz silnie spójne składowe w tym grafie.
3. Znajdź minimalne składowe.
4. Wyznacz najtańszych agentów w minimalnych składowych.
5. Zsumuj ceny najtańszych agentów w minimalnych składowych.
6. Wypisz wynik.

W implementacji tego algorytmu w programie AGE.PAS korzystamy z następujących struktur danych:

```
type
 TNrAgenta=0..CMAXAGENTOW;
 TWaluta  =1..CNIEPRZEKUPNY;
 TPapiery =0..CMAXPAPIEROW;
 PSasiad  =^TSasiad;
 TSasiad  =record
             Agent:TNrAgenta;
             Nast :PSasiad
            end;
 TAgent   =record
             MamPapiery  :PSasiad;
             MajaPapiery :PSasiad;
             Lapowka     :TWaluta;
             Numer       :TNrAgenta;
             NrSkladowej :TNrAgenta;
             RozPoddrzewa:TNrAgenta
            end;
 TSkladowa=record
             Minimalna   :boolean;
             MinLapowka  :TWaluta
            end;
var
 Agenci        :array[TNrAgenta] of TAgent;
 Numeracja     :array[TNrAgenta] of TNrAgenta;
 Skladowe      :array[TNrAgenta] of TSkladowa;
 IluAgentow    :TNrAgenta;
 IleSkladowych:TNrAgenta;
```

Stałe CMAXAGENTOW, CNIEPRZEKUPNY, CMAXPAPIEROW są równe odpowiednio: maksymalnej możliwej liczbie agentów, liczbie o 1 większej od największej możliwej przekupności agenta (oznaczającej, że agent jest nieprzekupny), maksymalnej możliwej liczbie dokumentów, jakie jedni agenci mogą mieć na temat innych agentów. Graf zależności między agentami jest pamiętany za pomocą list sąsiedztwa. Typ PSasiad reprezentuje jednokierunkową listę sąsiedzwa, a typ TSasiad jest typem elementów takiej listy. TAgent jest rekordem zawierającym informacje o pojedynczym agencie, na które składają się:

- MamPapiery – lista agentów, na temat których dany agent posiada dokumenty, czyli lista wierzchołków sąsiednich z danego wierzchołka w grafie zależności,

- `MajaPapiery` – lista agentów, którzy mają dokumenty na temat danego agenta, czyli lista wierzchołków, z których dany wierzchołek jest sąsiedni w grafie zależności;
- `Lapowka` – kwota, za którą można przekupić agenta (równa się `CNIEPRZEKUPNY`, gdy agent nie jest przekupny);
- `Numer` – numer agenta w numeracji, którą opisujemy poniżej;
- `NrSkladowej` – numer silnie spójnej składowej, do której należy dany agent;
- `RozPoddrzewa` – rozmiar poddrzewa rozpinającego graf, którego korzeniem jest dany agent – zobacz poniżej.

Typ `TSkladowa` zawiera informacje na temat jednej silnie spójnej składowej: znacznik wskazujący, czy jest ona minimalna, oraz minimalną kwotę, za którą można przekupić któregoś agenta w tej składowej (wynosi ona `CNIEPRZEKUPNY`, gdy wszyscy agenci w składowej są nieprzekupni). Tablica `Agenci` zawiera informacje na temat wszystkich agentów. Tablica `Numeracja` umożliwia odnalezienie agentów w tablicy `Agenci` zgodnie z ich numeracją pamiętaną w polach `Numer`. Tablica `Skladowe` zawiera informacje na temat wszystkich silnie spójnych składowych. Zmienne `IluAgentow` i `IleSkladowych` są równe liczbie wszystkich agentów oraz liczbie silnie spójnych składowych.

Najbardziej złożoną częścią rozwiązania jest wyznaczenie silnie spójnych składowych w grafie zależności między agentami. Pomysł algorytmu, który został zaczerpnięty z [6], opiera się na dwukrotnym, rekurencyjnym przeszukiwaniu grafu w głąb – raz zgodnie z kierunkiem łuków, a raz – przeciwnie do ich kierunku. Procedura `NumDFS` rekurencyjnie odwiedza wszystkich agentów, do których można dojść od agenta `Akt` przekazanego jako parametr. W procedurze `Numeruj` są odwiedzane wszystkie wierzchołki grafu.

```
procedure Numeruj;
 var AktNumer,i:TNrAgenta;
 procedure NumDFS(Akt:TNrAgenta);
  var Suwak:PSasiad;
 begin
  with Agenci[Akt] do begin
  Numer:=AktNumer;
  Numeracja[Numer]:=Akt;
  inc(AktNumer);
  RozPoddrzewa:=1;
  Suwak:=MamPapiery;
  while Suwak<>nil do begin
```

```
    if Agenci[Suwak^.Agent].Numer=CPUSTE then begin
     NumDFS(Suwak^.Agent);
     RozPoddrzewa:=RozPoddrzewa+Agenci[Suwak^.Agent].RozPoddrzew
    end;
    Suwak:=Suwak^.Nast
   end {while Suwak<>nil}
  end {with Agenci[Akt]}
end; {NumDFS}
begin
 AktNumer:=1;
 for i:=1 to IluAgentow do
  if Agenci[i].Numer=CPUSTE then NumDFS(i)
end; {Numeruj}
```

Odwiedzani agenci są numerowani w kolejności odwiedzania. W polu `Numer` jest pamiętany numer danego agenta, a w tablicy `Numeracja` jest pamiętana pozycja w tablicy `Agenci` agenta o danym numerze.

Zauważmy, że kolejne wywołania rekurencyjne odwiedzające agentów utworzą drzewo (a w ogólności – las, czyli zbiór poddrzew grafu zależności). Jest to **drzewo (las) rozpinające w głąb**. Odwiedzając agenta, procedura `NumDFS` zlicza również ilu agentów odwiedziła „poprzez" tego agenta (pole `RozPoddrzewa`) – jest to rozmiar fragmentu drzewa rozpinającego, którego korzeniem jest dany agent. Zauważmy, że jeżeli agent uzyskał numer x i „poprzez niego" zostało odwiedzonych k agentów, to mają oni numery $x, \ldots, x + k - 1$.

Zastanówmy się, jaki jest związek między drzewami rozpinającymi w głąb, a silnie spójnymi składowymi. Rozpoczynając przechodzenie drzewa od danego agenta odwiedzimy wszystkich agentów, do których możemy od niego dojść. W ten sposób na pewno odwiedzimy silnie spójną składową, do której on należy, ale możemy również odwiedzić niektóre inne składowe. Nie są one jednak ułożone dowolnie w odwiedzanym drzewie. Wszystkie wierzchołki danej składowej odwiedzimy „poprzez" ten wierzchołek składowej, który zostanie odwiedzony pierwszy. Inaczej mówiąc, silnie spójne składowe układają się w drzewie rozpinającym na kształt drzewa. Aby w obrębie drzewa rozpinającego znaleźć składową zawierającą korzeń drzewa, należy ponownie przeszukać graf – tym razem w kierunku przeciwnym do kierunku łuków, ale ograniczając się jedynie do agentów należących do wyznaczonego poprzednio drzewa. W ten sposób znajdziemy tych agentów, do których można dojść od korzenia drzewa i od których można wrócić do korzenia drzewa, a więc silnie spójną składową zawierającą korzeń. Podobnie postępujemy z pozostałymi składowymi znajdującymi się

w drzewie. Ważne jest jednak, aby wyznaczać składowe, do których należą agenci, w kolejności ustalonej poprzednio numeracji. W ten sposób korzeniem będzie zawsze ten wierzchołek w składowej, który pierwszy został odwiedzony. Oto odpowiedni fragment programu:

```
procedure SpojneSkladowe;
 var i:TNrAgenta;
 procedure SkladowaDFS(Akt:TNrAgenta);
  var Suwak:PSasiad;
 begin
  with Agenci[Akt] do begin
   NrSkladowej:=IleSkladowych;
   Suwak:=MajaPapiery;
   while Suwak<>nil do
    with Agenci[Suwak^.Agent] do begin
     if (NrSkladowej=CPUSTE) and (Numer>i) and
        (Numer<i+Agenci[Numeracja[i]].RozPoddrzewa)
     then SkladowaDFS(Suwak^.Agent);
     Suwak:=Suwak^.Nast
    end {with }
  end {with Agenci[Akt] }
 end; {SkaldowaDFS}
begin
 IleSkladowych:=0;
 for i:=1 to IluAgentow do
  if Agenci[Numeracja[i]].NrSkladowej=CPUSTE then begin
   inc(IleSkladowych);
   SkladowaDFS(Numeracja[i])
  end
end; {SpojneSkladowe}
```

Zauważmy, że przechodząc graf zależności między agentami, zarówno w kierunku łuków, jak i w kierunku przeciwnym, każdego agenta odwiedzamy tylko raz i każdy łuk wykorzystujemy też tylko raz. Tak więc wyznaczanie silnie spójnych składowych będzie trwało wprost proporcjonalnie długo do rozmiaru danych wejściowych – $O(n + m)$, gdzie n to liczba agentów, a m, to liczba łuków w grafie zależności.

Drugą fazę opisanego algorytmu możemy trochę przyspieszyć. Interesują nas tylko minimalne spójne składowe. Zauważmy, że w każdym drzewie rozpina-

jącym (zgodnie z kierunkiem łuków), tylko jedna składowa może być minimalna – ta, do której należy korzeń. Tak więc, od razu po zakończeniu przeszukiwania zgodnie z kierunkiem łuków, można wykonać przeszukiwanie w kierunku przeciwnym, zaczynając od tego samego agenta. Wszystkich agentów, dla których nie wyznaczymy ich składowych, można umieścić w jednej składowej, o której wiemy, że nie jest minimalna.

Aby wyznaczyć minimalne silnie spójne składowe wystarczy przejrzeć wszystkie łuki, wybrać te, które łączą różne składowe i zaznaczyć te składowe, do których nie prowadzi żaden łuk.

```
procedure MinSkladowe;
 var i:TNrAgenta; Suwak:PSasiad;
begin
 for i:=1 to IluAgentow do
  with Agenci[i] do begin
   Suwak:=MamPapiery;
   while Suwak<>nil do begin
    if NrSkladowej<>Agenci[Suwak^.Agent].NrSkladowej then
     Skladowe[Agenci[Suwak^.Agent].NrSkladowej].Minimalna:=
                false;
    Suwak:=Suwak^.Nast
   end {while}
 end {with, for}
end; {MinSkladowe}
```

Aby w (minimalnych) składowych wyznaczyć najtańszych agentów, wystarczy przejrzeć wszystkich agentów. Szukaną kwotę wyznaczamy przeglądając wszystkie składowe i sumując koszty przekupienia najtańszych agentów w minimalnych składowych. Gdyby w którejś z minimalnych składowych wszyscy agenci byli nieprzekupni, wystarczy w rozwiązaniu podać któregokolwiek agenta z tej składowej, jako przykład agenta, którego nie da się ani przekupić, ani aresztować.

Rozwiązanie prawie optymalne

Przedstawimy teraz krótko rozwiązanie opierające się na „błądzeniu" po grafie zależności. Zaczynając od dowolnego agenta, przechodzimy od agenta do agenta, ale zawsze w kierunku przeciwnym do kierunku łuków. Wcześniej czy później musi zajść jedna z dwóch sytuacji: możemy natrafić na agenta, do którego nie wchodzi żaden łuk, lub możemy dojść do agenta, którego już odwiedziliśmy. W obydwu przypadkach redukujemy problem (i graf) do mniejszego:

1. Jeśli do agenta nie wchodzi żaden łuk, to jest to agent stanowiący jednoelementową, minimalną, silnie spójną składową. Jeśli nie jest on przekupny, to odpowiedź w rozwiązaniu brzmi NIE. Jeśli jest przekupny, to musimy go przekupić – możemy wtedy go usunąć wraz ze wszystkimi agentami, których można dzięki niemu aresztować (stosujemy przeszukiwanie w głąb) i zająć się pozostałymi agentami.
2. Jeśli wrócimy do agenta, którego już odwiedziliśmy, to zamkniemy pewien cykl. Wszyscy agenci w tym cyklu muszą należeć do jednej silnie spójnej składowej grafu. Możemy więc ich „skleić" w jednego agenta, któremu przypiszemy minimalną przekupność spośród nich. Po sklejeniu cyklu możemy kontynuować błądzenie.

Powtarzając takie postępowanie albo uzyskamy listę agentów, których musimy przekupić, albo uzyskamy odpowiedź NIE oraz agenta, którego nie da się ani przekupić, ani aresztować.

Kluczowa dla efektywności tego rozwiązania jest realizacja operacji „sklejania" agentów. Należy zwrócić uwagę na dwie rzeczy: sklejanie list sąsiedztwa oraz sklejanie numerów agentów. Do sklejania numerów agentów można użyć dodatkowej struktury danych, w której dla każdego agenta będziemy pamiętać numer agenta, do którego został on „doklejony" (lub jego własny numer, jeśli nie został do nikogo doklejony). Wówczas, za każdym razem, gdy sięgamy do tablicy agentów, musimy najpierw sprawdzić w tej strukturze, do którego agenta dany agent został doklejony. Dobrą strukturą jest w tym przypadku *find-union* – operacja sklejania dwóch agentów nie trwa wówczas (średnio) dłużej niż $O(\log^* n)$ – zob. szczegóły w [3].

Sklejanie list sąsiedztwa wymaga więcej uwagi. W momencie sklejania agentów, ich listy sąsiedztwa sklejamy w jedną listę, bez przeglądania elementów list. Dwie listy można skleić w czasie stałym, więc będzie to trwało tyle, ilu agentów sklejamy. W ten sposób powstała lista może zawierać powtórzenia oraz łuki idące od sklejonego agenta do niego samego. Nie należy się tym jednak przejmować. Jedynie w trakcie dalszego błądzenia należy usuwać (napotkane) łuki prowadzące „do siebie samego". Umówmy się przy tym, że przy sklejaniu wierzchołków koszt związany z odwiedzeniem agenta, z którym sklejani są pozostali agenci w cyklu, bierze na siebie jeden z nich – w sposób nieformalny używamy tutaj techniki zwanej **kosztem zamortyzowanym** – zob. [6]. Możemy więc w dalszej części pominąć operacje związane z agentem, do którego są doklejani pozostali.

Prześledźmy los pojedynczego agenta. Może on zostać odwiedzony (co najwyżej dwa razy – gdy jest to agent zamykający cykl) w trakcie błądzenia, a następnie może zostać sklejony z innym agentem (biorąc – być może – na siebie

koszt odwiedzenia tego agenta) lub może zostać aresztowany i usunięty z grafu. Tak więc z każdym agentem wiąże się liczba czynności ograniczona przez stałą. Podobnie jest z łukami w grafie zależności. Błądząc możemy przejść przez każdy łuk co najwyżej raz – idąc wzdłuż cyklu lub po drodze do agenta minimalnego. Wyszukując aresztowanych agentów możemy przejść po każdym łuku co najwyżej raz. Każdy łuk zostanie usunięty – albo jako prowadzący od agenta do niego samego, albo jako wychodzący z aresztowanego agenta. Łączny koszt czasowy wynosi zatem $O(n\log^* n + m)$, czyli jest to algorytm prawie optymalny. Opisana wyżej realizacja algorytmu jest efektywna, lecz dość skomplikowana.

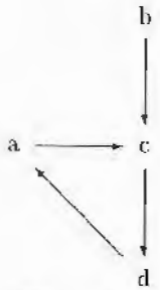
Omówienie rozwiązań podanych przez uczniów

Rozwiązanie zachłanne

Wśród rozwiązań uczniowskich znalazło się eleganckie, choć znacznie mniej efektywne rozwiązanie. Było to rozwiązanie zachłanne wykorzystujące przedstawione powyżej przeszukiwanie w głąb. (**Algorytm zachłanny** charakteryzuje się tym, że w kolejnych krokach rozszerza budowane rozwiązanie o optymalny w danej chwili element – więcej informacji na temat algorytmów zachłannych można znaleźć w [6].) Po wczytaniu danych i skonstruowaniu grafu zależności agenci są porządkowani według rosnących kwot, za które można ich przekupić. Agentów nieprzekupnych ustawiamy na końcu. Następnie, w tejże kolejności, przeglądamy agentów zaznaczając, których chcemy przekupić oraz których udało się dzięki temu aresztować. Jeśli kolejny agent jest przekupny, a nie został jeszcze aresztowany, to zaznaczamy, że go przekupujemy oraz przeszukujemy graf zaczynając od niego, zaznaczając wszystkich odwiedzanych agentów jako aresztowanych. Gdy w trakcie przeglądania grafu odwiedzimy agenta, którego wcześniej zaznaczyliśmy jako przekupionego, to wymazujemy to zaznaczenie. Następnie, po przejrzeniu wszystkich agentów, jeśli pozostał jakiś, który nie jest zaznaczony ani jako przekupiony, ani jako aresztowany – to rozwiązanie brzmi NIE. Jeśli wszyscy agenci są zaznaczeni, to wystarczy zsumować ceny agentów, których przekupujemy, aby uzyskać szukaną sumę.

W implementacji tego algorytmu struktury danych są prawie takie same jak w rozwiązaniu optymalnym, a procedura przeszukująca graf różni się jedynie tym, że zamiast numerować agentów zaznacza agentów aresztowanych.

Aby ten algorytm był poprawny, kolejne wywołania procedury przeszukującej graf muszą odwiedzić wszystkich agentów, do których można dojść od agenta przekazanego jako argument, bez względu na to, czy zostali oni już aresztowani, czy nie. Ilustruje to przykład na rys. 2.



Rysunek 2.

Niech agent a będzie przekupny za 10 zł, a agent b – za 20 zł. Algorytm zachłanny przekupi agenta a oraz aresztuje agentów c i d. Następnie przekupi agenta b. Aby jednak wycofać się z przekupywania agenta a musimy ponownie odwiedzić agentów c i d.

Ze względu na możliwość wielokrotnego odwiedzania tych samych agentów algorytm zachłanny działa w czasie $O(n(n+m))$, gdzie n jest liczbą agentów, a m jest liczbą krawędzi w grafie zależności.

Inne rozwiązania

Dosyć powszechną przyczyną błędów w rozwiązaniach uczniowskich były różne błędy implementacyjne. Znaczna część pozostałych błędów brała się z mylnego założenia, że zależności między agentami tworzą relację częściowego porządku. Poniżej przedstawiamy kilka najczęściej występujących błędów tego typu.

Niektóre rozwiązania polegały na przekupywaniu agentów, o których żaden z pozostałych agentów nie miał dokumentów. Oczywiście takich agentów należało przekupić – stanowią oni jednoelementowe, minimalne, silnie spójne składowe. Nie musieli to być jednak wszyscy agenci, których należało przekupić. Graf zależności może zawierać minimalne składowe składające się z większej liczby agentów.

Podobnie, część rozwiązań zachłannych opierała się na założeniu, że przekupienie agenta nie może doprowadzić do aresztowania żadnego agenta posiadającego dokumenty na jego temat. Programy realizujące takie rozwiązanie w trakcie przeszukiwania grafu odwiedzały każdego agenta tylko raz. Kontrprzykład, dla którego te rozwiązania działają błędnie podaliśmy na rys. 2.

Rozwiązania zachłanne, które nie sortowały agentów według ich cen, opierały się na założeniu, że w jednej silnie spójnej składowej nie może być dwóch przekupnych (za różne kwoty) agentów. Dla danych nie spełniających tego założenia mogły one działać błędnie (o ile w pewnej minimalnej składowej agent

przekupny za większą kwotę pieniędzy był rozpatrywany wcześniej niż agent przekupny za mniejszą kwotę).

Testy

Testy, za pomocą których oceniano rozwiązania zadania, można podzielić na dwie grupy: poprawnościowe i wydajnościowe. Testy poprawnościowe są zwykle niewielkie i sprawdzają, czy rozwiązanie działa poprawnie dla danych określonego rodzaju. Część z nich to tzw. „przypadki graniczne" (ekstremalne). Testy wydajnościowe sprawdzają efektywność rozwiązania – w przypadku tego zadania były to dosyć regularne grafy zależności między agentami o różnej wielkości. Testy wydajnościowe sprawdzają również poprawność rozwiązań, jednak jest to raczej sprawdzenie poprawności implementacji dla dużych danych, niż sprawdzenie poprawności algorytmu.

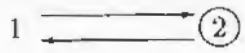
Programy uczniów miały ograniczony czas na wyliczenie odpowiedzi – po 5 sekund dla każdego z testów. Przekroczenie tego czasu oznaczało brak punktów za dany test.

Dla każdego testu przedstawiamy na rysunku graf zależności między agentami. Agentów przekupnych umieściliśmy w kółkach. Testy są prezentowane tematycznie, a więc czasem niezgodnie z ich numeracją.

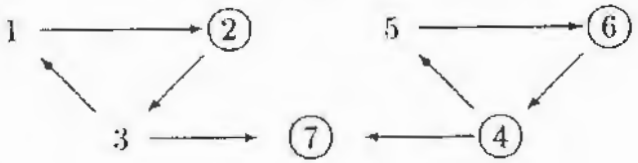
Test AGE0.IN jest pierwszym przykładem z treści zadania.

W teście AGE2.IN, agent nr 2 jest przekupny za 512 zł. Jest to przypadek graniczny – graf składa się tylko z jednej silnie spójnej składowej, zawierającej dwóch agentów. Należy przekupić agenta nr 2.

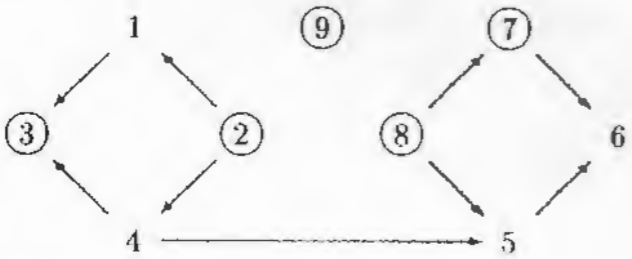
W teście AGE3.IN, agent nr 2 jest przekupny za 70 zł, nr 4 – za 10 zł, nr 6 – za 120 zł, a nr 7 – za 5 zł. Należy przekupić agentów nr 2 i 4. Obydwie mimimalne składowe zawierają więcej niż jednego agenta. W jednej z nich jest więcej niż jeden przekupny agent. Jest też agent przekupny (nr 7) poza minimalnymi składowymi.



Test AGE2.IN

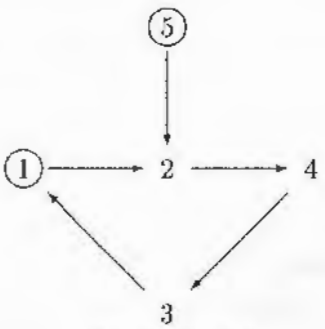


Test AGE3.IN



Test AGE4.IN

W teście AGE4.IN, agent nr 2 jest przekupny za 100 zł, nr 3 – za 10 zł, nr 7 – za 50 zł, nr 8 – za 1 000 zł, a nr 9 – za 10 000 zł. Jest to test eliminujący rozwiązania zachłanne, które nie eliminują raz przekupionych agentów. W teście tym należy przekupić najdroższych agentów, nr: 2, 8 i 9. Jest to też przypadek graniczny. Agent nr 9 nie jest powiązany z innymi agentami – stanowi jednoelementową silnie spójną składową.

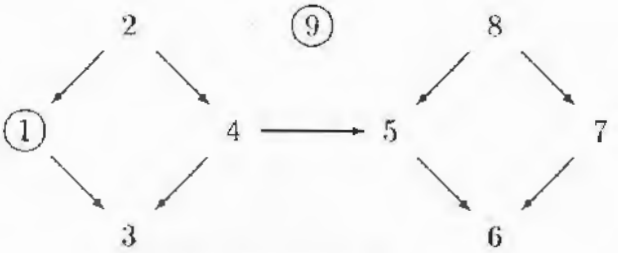


Test AGE12.IN

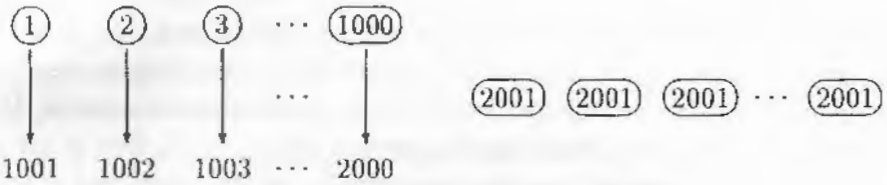
W teście AGE12.IN, dwaj agenci są przekupni: nr 1 za 10 zł i nr 5 za 20 zł. Jest to test eliminujący rozwiązania zachłanne, w których program nie odwiedza wielokrotnie tych samych agentów. Przekupienie agenta nr 1 powoduje aresztowanie agentów nr: 2, 4 i 3. Następnie, po przekupieniu agenta nr 5, należy ponownie odwiedzić/aresztować agentów nr: 2, 4 i 3, aby dojść do agenta nr 1 i aresztować go (zamiast przekupywać).

W teście AGE5.IN, przekupni są dwaj agenci: nr 1 za 100 zł oraz nr 9 za 10 000 zł. Jest to test na odpowiedź NIE. Nie da się ani przekupić, ani aresztować agentów nr: 2, 4, 5, 6, 7 oraz 8.

W teście AGE6.IN, agenci 1, ..., 1 000 są przekupni za 20 000 zł, a 2001, ..., 3000 – za 10000 zł. Jest to test badający efektywność: przeszukiwania



Test AGE5.IN

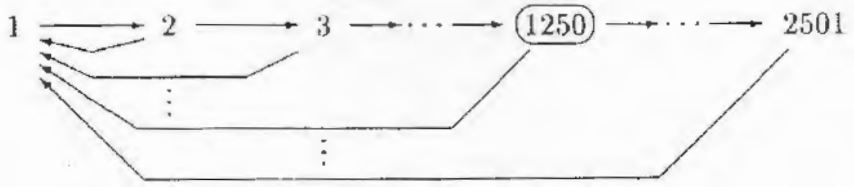


Test AGE6.IN

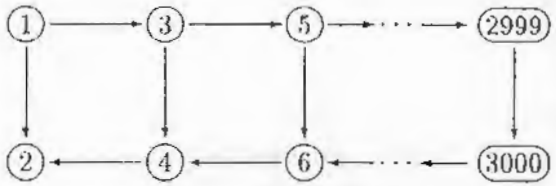
w głąb grafu zależności, czas potrzebny do znalezienia kolejnego agenta, którego trzeba przekupić oraz reprezentację arytmetyczną sumy pieniędzy potrzebnej do przekupienia agentów. Należy przekupić wszystkich przekupnych agentów, za każdym razem odwiedzając tylko jednego lub dwóch agentów. Test AGE10.IN jest analogiczny do testu AGE6.IN.

W teście AGE11.IN, przekupny jest tylko agent nr 1250 za 100 zł. Test ten bada efektywność rozwiązań sklejających agentów. Graf zależności między agentami składa się z jednej silnie spójnej składowej, ale zawiera wiele cykli, które mogą być sklejane. Test AGE1.IN jest analogiczny do testu AGE11.IN.

W teście AGE8.IN, wszyscy agenci są przekupni – agent nr i za $5i$ zł (dla $i = 1, \ldots, 3\,000$). Należy przekupić agenta nr 1. Ten test bada efektywność rozwiązań sklejających agentów. W teście AGE9.IN, w porównaniu z testem AGE8.IN, kierunek łuków między parami kolejnych agentów, z wyjątkiem ostatniej, jest przeciwny.

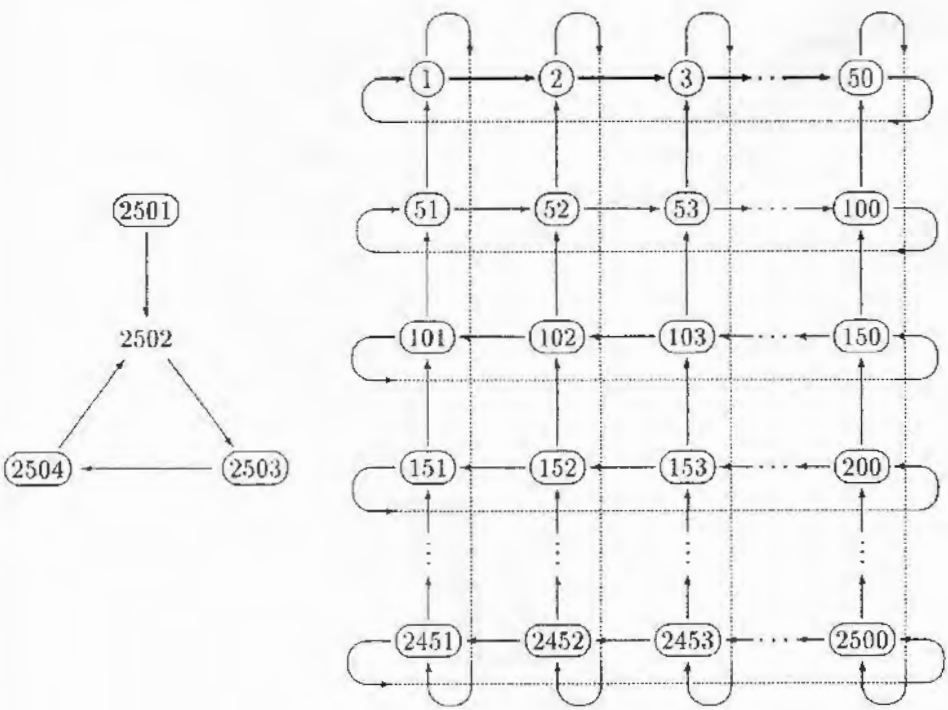


Test AGE11.IN



Test AGE8.IN

W teście AGE7.IN, wszyscy agenci, poza 2502 są przekupni: dla $i = 1, \ldots, 2500$, agent nr i jest przekupny za $2i$ zł, agent nr 2501 jest przekupny za 20 zł, agent nr 2503 – za 5 zł, a agent nr 2504 – za 10 zł. Ten test składa się z dwóch części: wydajnościowej i poprawnościowej. Część poprawnościowa eliminuje wadliwe rozwiązania zachłanne – podobnie jak niektóre z testów przedstawionych wyżej. Część wydajnościowa ma kształt zaburzonego (kierunki łuków) torusa. Bada ona efektywność rozwiązań sklejających wierzchołki. W teście tym należy przekupić agentów nr 1 i 2501.



Test AGE7.IN

7.3.3. Zadanie KULE (Autor: Krzysztof Diks)

Treść zadania KULE

Na okręgu umieszczono n pudełek ponumerowanych zgodnie z ruchem wskazówek zegara od 1 do n, gdzie $1 \leq n \leq 1000$. W pudełkach znajdują się kule, przy czym łącznie we wszystkich pudełkach jest ich nie więcej niż n.

Należy przełożyć kule w taki sposób, żeby w każdym pudełku pozostała co najwyżej jedna kula.

W jednym ruchu można przełożyć jedną kulę z pudełka, w którym się znajduje, do pudełka sąsiedniego.

Zadanie

Ułóż program, który:

- wczytuje z pliku tekstowego KUL.IN liczbę pudełek n i rozmieszczenie kul w pudełkach,
- oblicza minimalną liczbę ruchów, jakie trzeba wykonać, aby w każdym pudełku była co najwyżej jedna kula,
- zapisuje tę liczbę w pliku tekstowym KUL.OUT.

Wejście

W pierwszym wierszu pliku tekstowego KUL.IN jest zapisana jedna liczba całkowita dodatnia. Jest to liczba pudełek n.

W każdym z kolejnych n wierszy jest zapisana jedna liczba całkowita nieujemna. W i-tym z tych wierszy jest zapisana liczba kul w i-tym pudełku.

Wyjście

W pliku tekstowym KUL.OUT należy zapisać jedną liczbę całkowitą nieujemną, tj. minimalną liczbę ruchów, jakie trzeba wykonać, aby w każdym pudełku była co najwyżej jedna kula.

Przykład

Dla pliku KUL.IN:

```
12
0
0
2
4
3
1
0
```

0
0
0
0
1

w pliku KUL.OUT należy zapisać:

19

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

Rozwiązania zadania KULE

Przedstawimy dwa rozwiązania tego zadania. Pierwsze rozwiązanie będzie działało w czasie $O(n^3)$, a drugie – w czasie $O(n^2)$. Program KUL.PAS jest realizacją tego drugiego rozwiązania.

W opisach obu rozwiązań używamy tych samych oznaczeń i definicji. Pudełka identyfikujemy z ich numerami. O kulach zakładamy, że są poetykietowane. **Etykietą** kuli jest numer pudełka zawierającego początkowo tę kulę. **Początkowym rozmieszczeniem** kul nazywamy funkcję $P : \{1, 2, \ldots, n\} \to \{0, 1, \ldots, n\}$ taką, że $\sum_{j=1}^{n} P(j) \le n$, gdzie $P(i)$ jest początkową liczbą kul w i-tym pudełku, a zatem $\sum_{j=1}^{n} P(j)$ jest liczbą wszystkich kul w pudełkach – oznaczmy tę ostatnią liczbę przez m. **Rozmieszczeniem końcowym** kul dla początkowego rozmieszczenia P nazywamy funkcję $K : \{1, 2, \ldots, n\} \to \{0, 1, \ldots, n\}$ spełniającą następujące dwa warunki:

1. Dla każdego $i = 1, \ldots, n$, liczba tych j, dla których $K(j) = i$, jest równa $P(i)$.
2. Liczba tych j, dla których $K(j) > 0$, jest równa m.

Innymi słowy, funkcja K podaje etykiety kul, które w rozmieszczeniu końcowym wpadają do poszczególnych pudełek. Jeżeli $K(i) = 0$, to w rozmieszczeniu końcowym w i-tym pudełku nie ma żadnej kuli. Warunek 1 oznacza, że w rozmieszczeniu końcowym kul o etykiecie i jest dokładnie tyle, ile początkowo było ich w pudełku i-tym. Warunek 2 gwarantuje, że rozmieszczone zostały tylko te kule, które na początku znajdowały się w pudełkach.

Koszt umieszczenia kuli w i-tym pudełku jest równy $c(i)$, gdzie:

$$c(i) = \begin{cases} 0 & K(i) = 0, \\ \min(|K(i) - i|, n - |K(i) - i|), & K(i) > 0. \end{cases}$$

Zatem, koszt $c(i)$ jest równy liczbie ruchów, które trzeba wykonać, aby przenieść kulę z pudełka o numerze $K(i)$ do pudełka o numerze i. **Kosztem** rozmieszczenia końcowego K jest zatem wartość

$$C(K) = \sum_{i=1}^{n} c(i).$$

Rozmieszczenie końcowe K jest **optymalne** dla rozmieszczenia początkowego P, jeżeli koszt rozmieszczenia K nie jest większy od kosztu jakiegokolwiek innego rozmieszczenia końcowego dla rozmieszczenia początkowego P. Koszt $C(K)$ jest wtedy **minimalny** dla rozmieszczenia początkowego P i oznaczamy go przez $MinC(P)$. Naszym celem jest podanie efektywnego algorytmu (a następnie napisanie programu), który dla danego początkowego rozmieszczenia kul P oblicza $MinC(P)$.

Wyprowadzenie algorytmu rozpoczniemy od poczynienia pewnych spostrzeżeń dotyczących rozmieszczeń optymalnych. Niech K będzie rozmieszczeniem końcowym dla P. Maksymalny ciąg kolejnych niepustych pudełek, tzn. taki, którego nie można rozszerzyć ani z lewej (przeciwnie do ruchu wskazówek zegara), ani z prawej strony (zgodnie z ruchem wskazówek zegara), zawierający kule o takiej samej etykiecie (czyli pochodzące z tego samego pudełka) nazywamy **blokiem**. Maksymalny ciąg niepustych pudełek nazywamy **superblokiem**. Superblok może zawierać więcej niż jeden blok. **Etykietą bloku** jest etykieta zawartych w nim kul.

Dla pary kolejnych pudełek $i, i+1$ (za $i+1$ bierzemy 1, gdy $i=n$) mówimy, że został wykonany **ruch w prawo**, jeżeli pewna kula została przemieszczona w kierunku zgodnym z ruchem wskazówek zegara z pudełka na lewo od $i+1$ do pudełka na prawo od i. (Uwaga: Jeżeli odległości między pudełkami a i b są takie same w kierunku zgodnym z ruchem wskazówek zegara i w kierunku przeciwnym do ruchu wskazówek zegara, to kulę z a do b przemieszczamy w kierunku zgodnym z ruchem wskazówek zegara.) Podobnie mówimy, że dla pary $i, i+1$ został wykonany **ruch w lewo**, jeżeli pewna kula została przemieszczona w kierunku przeciwnym do ruchu wskazówek zegara z pudełka na prawo od i do pudełka na lewo od $i+1$.

Spostrzeżenie 1. *Jeżeli K jest rozmieszczeniem optymalnym, to dla żadnej pary kolejnych pudełek nie mogą być jednocześnie wykonane ruchy w lewo i w prawo.*

Aby uzasadnić to spostrzeżenie, niech $i, i+1$ będzie parą pudełek, dla których wykonywany jest zarówno ruch w lewo, jak i ruch w prawo. Niech a będzie kulą wędrującą w lewo, a b będzie kulą wędrującą w prawo. Wówczas, jeżeli

zamienimy docelowe pudełka kul a i b, to uzyskamy rozmieszczenie o mniejszym koszcie, wbrew temu co założyliśmy o rozmieszczeniu K.

Spostrzeżenie 2. *Jeżeli K jest rozmieszczeniem optymalnym, to dla żadnego pudełka i nie może jednocześnie być wykonywany ruch w prawo dla pary $i-1, i$ (dla $i = 1$ przyjmujemy, że $i - 1 = n$) i w lewo dla pary $i, i+1$.*

Gdyby tak było, to ze spostrzeżenia 1 wynikałoby, że K nie może być rozmieszczeniem końcowym, gdyż pudełko i zawierałoby więcej niż jedną kulę.

Spostrzeżenie 3. *Jeżeli K jest rozmieszczeniem optymalnym, to istnieje pudełko i takie, że dla pary $i, i+1$ nie jest wykonywany ani ruch w lewo, ani ruch w prawo.*

Jeżeli dla każdego pudełka byłby wykonywany jakiś ruch, to wszystkie ruchy musiałyby być wykonane w jednym kierunku. Załóżmy, że jest to kierunek w prawo (zgodny z ruchem wskazówek zegara). Symetrycznie rozpatruje się ruchy w lewo. Bez utraty ogólności możemy założyć, że żadna kula nie pozostaje w swoim pudełku. To założenie można uzasadnić tym, że dla każdego i jest wykonywany ruch w prawo od $i-1$ do i. Stąd wynika, że jedną z kul przemieszczających się od $i-1$ do i można pozostawić w i, a w jej miejsce przesuwać dalej tę kulę z pudełka i, która uprzednio miała w nim pozostać. Taka zmiana nie wpływa na koszt rozmieszczenia. Teraz jednak łatwo zauważyć, że po przesunięciu wszystkich kul o jedno pudełko w lewo otrzymujemy rozmieszczenie o koszcie mniejszym od kosztu K, wbrew założeniu, że rozmieszczenie K jest optymalne.

Spostrzeżenie 3 umożliwia sprowadzenie zadania o przemieszczaniu kul dla pudełek na okręgu do zadania o przemieszczaniu kul dla pudełek umieszczonych na linii. W tym celu tworzymy n linii pudełek: $1, 2, \ldots, n$; $2, 3, \ldots, n, 1$; $3, 4, \ldots, n, 1, 2$; $\ldots; n, 1, 2, \ldots, n-1$ i dla każdej z nich znajdujemy koszt optymalnego rozmieszczenia kul. Najmniejszy z tych kosztów jest poszukiwanym minimalnym kosztem rozmieszczenia kul dla pudełek na okręgu. Na linii $p_1, p_2, \ldots, p_n$ jest tylko jedna możliwość przesunięcia kuli z pudełka p_i do pudełka p_j. Jeżeli $i < j$, to kula wędruje kolejno przez pudełka $p_i, p_{i+1}, p_{i+2}, \ldots, p_{j-1}, p_j$. Jeżeli natomiast $i > j$, to kula wędruje kolejno przez pudełka $p_i, p_{i-1}, p_{i-2}, \ldots, p_{j+1}, p_j$. Liczba wykonanych ruchów w tym przypadku wynosi $|i-j|$. Pokażemy, w jaki sposób rozwiązać zadanie dla linii w czasie $O(n^2)$, co daje dla naszego wyjściowego problemu algorytm działający w czasie $O(n^3)$.

Załóżmy, że mamy rozmieszczenie początkowe P dla linii $1, 2, \ldots, n$. Nasze rozwiązanie opiera się na następującym, łatwym do wykazania fakcie.

Spostrzeżenie 4. *Istnieje optymalne rozmieszcze końcowe kul na linii, w którym kule pochodzące z tego samego pudełka znajdują się w jednym bloku, a bloki występują na linii w kolejności rosnących etykiet, tzn. w takiej samej kolejności jak niepuste pudełka w rozmieszczeniu początkowym P. Ponadto, do każdego superbloku należą wszystkie pudełka, z których pochodzą kule umieszczone w jego blokach.*

Ponieważ optymalne rozmieszczenie kul na jednej z linii, powstającej w wyniku rozcięcia okręgu, jest optymalnym rozmieszczeniem kul dla pudełek na okręgu, powyższe spostrzeżenie odnosi się także do okręgu.

Dla znalezienia optymalnego rozmieszczenia kul na linii nadajemy kulom kolejne numery 1, 2, ..., poczynając od kul w pierwszym niepustym pudełku, a kończąc na ostatnim niepustym pudełku. Minimalny koszt rozmieszczenia kul na linii dla danego początkowego rozmieszczenia P obliczymy za pomocą **metody programowania dynamicznego**, inaczej nazywanej **metodą tablicowania kosztów**. Algorytm polega na stablicowaniu funkcji $C(q, p)$ określającej minimalny koszt rozmieszczenia q pierwszych kul w p pierwszych pudełkach i zdefiniowanej następująco:

$$C(0, p) = 0,$$

$$C(q, p) = \min\begin{cases} C(q-1, p-1) + d(pud(q), p), \\ C(q, p-1), \end{cases} \quad \text{jeżeli } p - 1 \geq q,$$

gdzie $pud(q)$ jest numerem pudełka, z którego pochodzi kula o numerze q, a $d(r, s)$ jest odległością pudełek r i s. (Uwaga: dla $p < q$ przyjmujemy, że $C(p, q) = \infty$.)

Naszym celem jest obliczenie $C(m, n)$. Aby to zrobić należy wypełnić wartościami tablicę $C[0..m, 1..n]$. Zauważmy, że dla obliczenia $C(q, p)$ wystarczy znać jedynie $C(q-1, p-1)$ i $C(q, p-1)$. Stąd wynika, że tablicowanie można wykonywać wierszami i wystarczy pamiętać tylko dwa ostatnio wiersze: q i $q-1$. Odległość $d(r, s)$ oblicza się w czasie $O(1)$. Funkcję $pud(q)$ można stablicować w czasie czytania danych. Z powyższych rozważań otrzymujemy, że koszt optymalnego rozmieszczenia kul dla linii można znaleźć w czasie $O(n^2)$ i w pamięci o rozmiarze $O(n)$. Stosując zaprezentowany algorytm dla n różnych linii powstających w wyniku rozcięcia okręgu pudełek, uzyskujemy algorytm rozwiązania oryginalnego zadania, który działa w czasie $O(n^3)$ i wykorzystuje pamięć o rozmiarze $O(n)$.

Przedstawione rozwiązanie jest mało efektywne szczególnie wtedy, gdy liczba pudełek n i liczba kul m są bliskie 1000. Przedstawimy teraz rozwiązanie równie proste, ale działające w czasie $O(n^2)$.

Nazwijmy **spójnym** optymalne rozmieszczenie końcowe, w którym kule pochodzące z tego samego pudełka znajdują się w jednym bloku, bloki występują na okręgu w takiej samej kolejności jak pudełka, z których pochodzą występujące w nich kule, a każdy superblok zawiera wszystkie pudełka zawierające występujące w nim kule. Ze spostrzeżenia 4 wiemy, że takie rozmieszczenie końcowe istnieje. Niech P będzie rozmieszczeniem początkowym, a K spójnym, optymalnym rozmieszczeniem końcowym dla P. Załóżmy też, że $m > 0$. Niech S będzie superblokiem złożonym z bloków, w kolejności od lewej do prawej, $B_1, B_2, \ldots, B_k$. Rozważmy blok B_i, dla dowolnego $i = 1, \ldots, k$. Niech l_i będzie skrajnie lewym pudełkiem w tym bloku. Zastanówmy się, w jaki sposób z rozmieszczenia K otrzymać spójne, optymalne końcowe rozmieszczenia K' dla początkowego rozmieszczenia P', powstającego z P przez usunięcie kuli znajdującej się w pudełku l_i w rozmieszczeniu K. Niech R będzie rozmieszczeniem powstającym z K przez usunięcie kuli z pudełka l_i. W wyniku usunięcia kuli z l_i mogą powstać dwa nowe superbloki: $S_1 = [B_1, \ldots, B_{i-1}]$ i $S_2 = [B_i', B_{i+1}, \ldots, B_k]$, gdzie B_i' jest blokiem B_i bez pudełka l_i. Blok B_i' może zostać zredukowany do bloku pustego i wówczas $S_2 = [B_{i+1}, \ldots, B_k]$. Może zajść jeden z dwóch przypadków.

1. R jest rozmieszczeniem optymalnym dla P'.
   Nietrudno zauważyć, że R jest rozmieszczeniem spójnym. Uzasadnienia wymaga jedynie, że superbloki zawierają pudełka, z których pochodzą kule występujące w ich składowych blokach.
   Jeżeli superblok S_2 zawiera pudełko, z którego pochodzą kule z bloku B_{i-1}, to przemieszczając skrajnie prawą kulę z B_{i-1} do pudełka l_i otrzymujemy rozmieszczenie lepsze od R, wbrew założeniu, że R jest optymalne. Podobnie, jeżeli superblok S_1 zawiera pudełko, z którego pochodzą kule z B_i' (lub B_{i+1}, jeśli B_i' jest pustym blokiem), to przemieszczając skrajnie lewą kulę z B_i' (B_{i+1}) do pudełka l_i, otrzymujemy rozmieszczenie lepsze od R, wbrew założeniu o optymalności tego rozmieszczenia.
   Zatem w tym przypadku za K' można przyjąć rozmieszczenie R.
2. R nie jest rozmieszczeniem optymalnym dla P'.
   Niech Q będzie dowolnym optymalnym, końcowym rozmieszczeniem dla P'. Zauważmy, że pudełko l_i nie może być w tym rozmieszczeniu puste. W przeciwnym razie rozmieszczenie Q razem z umieszczoną z powrotem w l_i usuniętą wcześniej kulą jest rozmieszczeniem o mniejszym koszcie niż K, co jest sprzeczne z założeniem, że K jest optymalne. Rozważmy ciąg pudełek $p_1 = l_i, p_2, \ldots, p_k$ taki, że przy przejściu od rozmieszczenia R do rozmieszczenia Q do pudełka p_i trafia kula z pudełka p_{i+1}, dla każdego $i = 1, 2, \ldots, k-1$, a pudełko p_k zostaje opróżnione. Bez utraty ogólności możemy założyć, że żadne inne kule nie były przemieszczame. Gdyby bowiem ruch

pozostałych kul miał wpływ na zmniejszenie kosztu rozmieszczenia, to wykonując te same przemieszczenia kul dla rozmieszczenia K dostalibyśmy rozmieszczenie o mniejszym koszcie. Zauważmy teraz, że żadne z pudełek p_i nie może należeć do superbloku innego niż S. Załóżmy przeciwnie. Niech p_j będzie pierwszym pudełkiem należącym do innego bloku niż S. Niech tym blokiem będzie S'. Wiemy, że kula przesuwana z pudełka p_j do pudełka p_{j-1} pochodzi z pudełka należącego do superbloku S', powiedzmy q. Wówczas, zamiast przenosić kulę z p_j do p_{j-1} lepiej przenieść ją z tego pudełka, do jednego z pustych pudełek sąsiadujących bezpośrednio z S' z lewej lub z prawej strony, a konkretnie tego, które leży bliżej q. Wykonując pozostałe ruchy bez zmian otrzymamy rozmieszczenie o koszcie mniejszym niż koszt Q – sprzeczność z założeniem o optymalności Q. Zatem wszystkie pudełka p_j należą do S. W wyniku przemieszczenia pudełek opróżnione zostaje pudełko p_k. Puste pudełko p_k dzieli superblok S na dwa superbloki L i P (któryś z nich może być pusty). Do L należą pudełka z S leżące na lewo od p_k, a do P należą pudełka z S leżące na prawo od P_k. Argumentując podobnie jak w 1 można pokazać, że w L są tylko te kule, które pochodzą z pudełek z L, a w P są tylko te kule, które pochodzą z pudełek z P. Nietrudno zauważyć, że w każdym superbloku L i P istnieje spójne, optymalne, końcowe rozmieszczenie. Te rozmieszczenia, plus niezmienione rozmieszczenia w superblokach innych niż S, wyznaczają optymalne, spójne, końcowe rozmieszczenie K' dla P'.

Jak otrzymaliśmy K' z K? Najpierw usunęliśmy skrajny element z bloku B_i. W wyniku tego, albo otrzymaliśmy rozmieszczenie optymalne, albo w celu otrzymania takiego rozmieszczenia należało przesunąć pewną liczbę bloków z S o jedną pozycję w prawo (są to bloki z lewej strony opróżnionego pudełka) lub należało przesunąć pewną liczbę bloków o jedną pozycję w prawo (bloki z prawej strony opróżnionego pudełka). Zawsze wykonujemy takie przesunięcia, które minimalizują końcowy koszt rozmieszczenia.

Powyższe rozważania prowadzą do następującego algorytmu obliczania spójnego, optymalnego, końcowego rozmieszczenia K dla początkowego rozmieszczenia P.

Na potrzeby tego algorytmu załóżmy, że kule są przenoszone po jednej ze starego kompletu pudełek, do nowego kompletu pustych pudełek. Na początku, do każdego nowego pudełka jest przenoszona jedna kula z odpowiadającego mu starego pudełka. Następnie przenosi się kolejne kule w ten sposób, że:

1. W każdym nowym pudełku zostaje umieszczona co najwyżej jedna kula.
2. Wszystkie nowe pudełka, w których są umieszczone kule z tego samego starego pudełka, tworzą na okręgu blok.

Przeniesienie jednej kuli ze starego pudełka p polega na znalezieniu bloku nowych pudełek zawierających kule z pudełka p oraz dołożeniu kuli z lewej lub prawej strony bloku i ewentualnym przesunięciu kul z sąsiednich bloków o jedno pudełko, jeżeli brak wolnego miejsca. Wybór (z której strony dołożyć) zależy od kosztu operacji, tj. kosztu umieszczenia kuli plus ewentualne zmiany kosztu rozmieszczenia innych kul, jeżeli zostały przesunięte. Zatem nową kulę dokładamy tak, aby otrzymać spójne, optymalne rozmieszczenie końcowe, a sposób w jaki to robimy, jest odwrotny do tego, który opisaliśmy dla usuwania kuli. Gwarantuje to, że końcowe rozmieszczenie będzie rzeczywiście optymalne.

Nową kulę można dostawić w czasie $O(n)$. Zatem czas działania całego algorytmu wynosi $O(n^2)$. Rozmiar pamięci użytej do implementacji powyższego algorytmu jest liniowy $O(n)$.

Szczegółowa implemetacja algorytmu znajduje się w programie KUL.PAS*.

Omówienie rozwiązań podanych przez uczniów

To zadanie okazało się zadaniem bardzo trudnym. Na 38 uczniów biorących udział w zawodach III stopnia, tylko jeden (Przemysław Gordinowicz) przedstawił w pełni zadowalające rozwiązanie i uzyskał prawie maksymalną liczbę punktów (46 na 50 możliwych), a drugie w kolejności rozwiązanie uzyskało 22 punkty. Najlepsze rozwiązanie uczniowskie było w pewnym sensie podobne do rozwiązania autorskiego. Polegało ono na poprawianiu bieżącego rozwiązania optymalnego przez dostawianie pojedynczych kul. Kryterium wyboru miejsca wstawienia nowej kuli było jednak inne niż powyżej. Dla każdej pary pudełek $i, i+1$ były pamiętane liczba i rodzaj ruchów (w lewo lub w prawo), które zostały już wykonane dla tej pary. Liczby te wyznaczały w sposób ukryty rozmieszczenie optymalne. Dostawianie było zawsze tak wykonywane, aby nowe rozmieszczenie (po umieszczeniu nowej kuli) było również optymalne. Koszt dostawienia nowej kuli był liniowy, zatem cały algorytm działał w czasie $O(n^2)$.

Rozwiązania innych uczniów polegały na wystartowaniu od „przybliżonego" rozmieszczenia końcowego (z co najwyżej jedną kulą w każdym pudełku) i poprawianiu go przez przemieszczanie kul w pudełkach. Wyszukiwanie kul, które należało przemieścić, było zazwyczaj dość kosztowne i dlatego te rozwiązania były nieefektywne.

---

* Autor rozwiązania (K. Diks) dziękuje Grzegorzowi Jakackiemu, który opracowywał rozwiązanie tego zadania w ramach prac Jury Olimpiady i pozwolił z niego skorzystać w tym opisie. Jest on również autorem programu KUL.PAS.

Część uczniów stosowała algorytm zachłanny, polegający na rozpatrywaniu kul pojedynczo i umieszczaniu każdej z nich w najbliższym pustym pudełku. Stosowano różne heurystyki do wyznaczania kolejności, w jakiej kule należało brać pod uwagę, ale niestety heurystyki te były niepoprawne.

Żaden z uczniów nie zauważył, że rozwiązanie zadania dla okręgu można sprowadzić do zadania dla linii i zastosować programowanie dynamiczne.

Testy

Do sprawdzenia rozwiązań użyto 19 testów KUL0.IN, ...,KUL18.IN. Test KUL18.IN był przykładem z treści zadania. Liczba pudełek wahała się od 12 do 1 000, a minimalna liczba ruchów od 0 do 214 438. Celem testów było sprawdzenie zarówno poprawności, jak i szybkości działania rozwiązań. Testy od KUL1.IN do KUL4.IN badały zachowanie się programów dla nierównomiernych rozkładów kul, natomiast testy od KUL12.IN do KUL15.IN sprawdzały działanie programów dla równomiernych rozkładów kul. Testy KUL16.IN i KUL17.IN sprawdzały, czy programy radzą sobie z przypadkiem, w którym rozmieszczenie początkowe jest jednocześnie rozmieszczeniem końcowym. Poprawność doboru typów liczbowych była badana między innymi za pomocą testów KUL5.IN i KUL15.IN. W pozostałych testach początkowe rozmieszczenia kul były losowe.

7.3.4. Zadanie NIE ATAKUJĄCE SIĘ SKOCZKI (Autor: Wojciech Rytter)

Treść zadania NIE ATAKUJĄCE SIĘ SKOCZKI

Skoczek jest figurą, która atakuje pola na szachownicy zgodnie z rysunkiem 1.

Dana jest szachownica o rozmiarach $3 \times n$ – mająca 3 wiersze i n kolumn, gdzie $1 \leq n \leq 100$ – oraz zbiór Z pól na tej szachownicy.

	X		X	
X				X
		S		
X				X
	X		X	

Rysunek 1. Skoczek umieszczony w polu S atakuje pola oznaczone przez X.

Wiersze na szachownicy są ponumerowane od góry do dołu liczbami od 1 do 3, a kolumny od lewej do prawej liczbami od 1 do n.

Na szachownicy wolno rozmieszczać skoczki tylko na polach nie należących do Z oraz tak, aby żadne dwa nie atakowały się nawzajem.

Zakładamy, że w każdej kolumnie co najwyżej jedno pole należy do Z. Zbiór Z można więc opisać w postaci ciągu: $k_1, k_2, \ldots, k_n$, gdzie $k_i \in \{0, 1, 2, 3\}$. Wartość $k_i = 0$ oznacza, że żadne pole w i-tej kolumnie nie należy do Z, zaś wartość $k_i > 0$ jest numerem wiersza jedynego pola w tej kolumnie, które należy do Z.

Zadanie

Ułóż program, który:

- wczytuje z pliku tekstowego NIE.IN liczbę kolumn na szachownicy n oraz opis zbioru pól Z,
- znajduje maksymalną liczbę skoczków M, które można ustawić na szachownicy zgodnie z podanymi zasadami oraz liczbę L wszystkich takich ustawień M skoczków,
- zapisuje wyniki w pliku tekstowym NIE.OUT.

Wejście

W pierwszym wierszu pliku tekstowego NIE.IN jest zapisana jedna liczba całkowita dodatnia $n \leq 100$. Jest to liczba kolumn szachownicy.

W każdym z kolejnych n wierszy jest zapisana jedna liczba ze zbioru {0, 1, 2, 3}. Są to kolejne wyrazy ciągu będącego opisem zbioru pól Z.

Wyjście

W pliku tekstowym NIE.OUT należy zapisać dwie liczby całkowite M i L oddzielone odstępem.

Przykład

Dla pliku NIE.IN:

```
2
1
0
```

w pliku NIE.OUT należy zapisać:

```
4 2
```

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

programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku NIE.EXE.

Rozwiązanie zadania NIE ATAKUJĄCE SIĘ SKOCZKI

Rozwiązanie zadania, polegające na bezpośrednim przeglądania wszystkich możliwych ustawień skoczków, prowadzi do programu działającego w czasie niewielomianowym od rozmiaru szachownicy. Podstawową własnością zadania z punktu widzenia efektywności, z której skorzystamy, jest możliwość jego podziału za pomocą małego separatora, składającego się z sześciu pól szachownicy umieszczonych w dwóch kolumnach.

Przy ustawianiu skoczków kolumnami od lewej do prawej można zauważyć, że dla wypełnienia kolejnej kolumny istotną rolę, ze względu na zasięg ruchów, odgrywa ustawienie skoczków tylko w dwóch poprzednich kolumnach. Załóżmy, że obliczyliśmy już maksymalną liczbę skoczków i liczbę maksymalnych ustawień dla każdego możliwego ustawienia skoczków w ostatnich dwóch kolumnach k-kolumnowego **prefiksu szachownicy**, czyli w k pierwszych kolumnach szachownicy. Na tej podstawie wyznaczymy te liczby dla każdego możliwego ustawienia skoczków w ostatnich dwóch kolumnach $(k+1)$-kolumnowego prefiksu szachownicy.

Zaczynamy od dwukolumnowego prefiksu i postępując iteracyjnie znajdziemy w ten sposób maksymalną liczbę skoczków i liczbę ich rozstawień dla szachownicy n-kolumnowej. Ze względów technicznych (dla ułatwienia inicjalizacji) rozpatrujem ustawienia na szachownicy $(n+2)$-elementowej, w której w pierwszych dwóch kolumnach nie ma w ogóle skoczków.

Ustawienie skoczków w ostatnich dwóch kolumnach nazywamy **konfiguracją** i pamiętamy w wektorzy x o sześciu współrzędnych, które mogą przyjmować wartości 1 lub 0 (w programie jest to wektor typu `boolean`). Pola w ostatnich dwóch kolumnach numerujemy z góry na dół, najpierw w pierwszej kolumnie a następnie w drugiej. Przymujemy, że x_i = (*true*), gdy na i-tym polu w ostatnich dwóch kolumnach stoi skoczek, a 0 (*false*) – w przeciwnym przypadku. Konfiguracja x jest poprawna względem k, gdy skoczki nie stoją na zabronionych polach oraz dwa skoczki nie atakują się nawzajem. Oznaczmy odpowiadający predykat przez *Poprawna*(x, k). Z definicji przyjmijmy, że *Poprawna*$(x, 2)$, czyli dla $k = 2$, zachodzi jedynie dla konfiguracji **pustej** x_0, tzn. dla takiej konfiguracji, w której nie ma skoczków w obu kolumnach. Oznaczmy zbiór wszystkich konfiguracji przez K. Zauważmy, że K ma 2^6=64 elementy, przy czym część z nich może być niepoprawna.

Załóżmy, że ustawieniem w ostatnich dwóch kolumnach jest x, oraz w dwóch pierwszych kolumnach nie ma skoczków i zdefiniujmy dwie wielkości:

Max(x, k) – maksymalna liczba skoczków w pierwszych k kolumnach,
Ile(x, k) – liczba ustawień maksymalnej liczby skoczków w pierwszych k kolumnach.

Na podstawie przyjętych założeń mamy: $Max(x_0, 2) = Ile(x_0, 2) = 0$.

Oznaczmy przez M i L odpowiednio maksymalną liczbę skoczków na szachownicy o n kolumnach oraz liczbę maksymalnych ustawień. Z dotychczasowych rozważań mamy:

$$M = \max\{Max(x, n+2): x \in K\},$$

$$L = \sum_{x \in P} Ile(x, n+2),$$

gdzie

$$P = \{x: Max(x, n+2) = M\}.$$

Opiszemy teraz w terminach teorii grafów algorytm obliczania wartości *Max*(x, k) oraz *Ile*(x, k). Z zadaniem związemy graf $G = (V, E)$, a dokładniej graf skierowany, którego zbiór wierzchołków V składa się z konfiguracji w poszczególnych kolumnach i dwa wierzchołki u i v są połączone łukiem, czyli skierowną krawędzią, co zapisujemy $u \rightarrow v$, jeśli konfiguracje są ze sobą odpowiednio związane. Dokładniej, konfiguracja x jest **poprzednikiem** konfiguracji y, gdy ustawienie skoczków w drugiej kolumnie konfiguracji x jest takie samo jak w pierwszej kolumnie konfiguracji y oraz dostawiając do x drugą kolumnę y otrzymamy ustawienie skoczków w trzech sąsiednich kolumnach takie, że żadne dwa skoczki się nie atakują. Oznaczmy odpowiadający temu predykat przez *Poprzednik*(x, y).

Możemy już teraz zdefiniować graf $G = (V, E)$ określając zbiór wierzchołków V i zbiór łuków E w następujący sposób:

$$V = \{(x, k): x \in K,\ Poprawna(x, k),\ 2 < k \leq n+2\} \cup \{(x_0, 2)\},$$

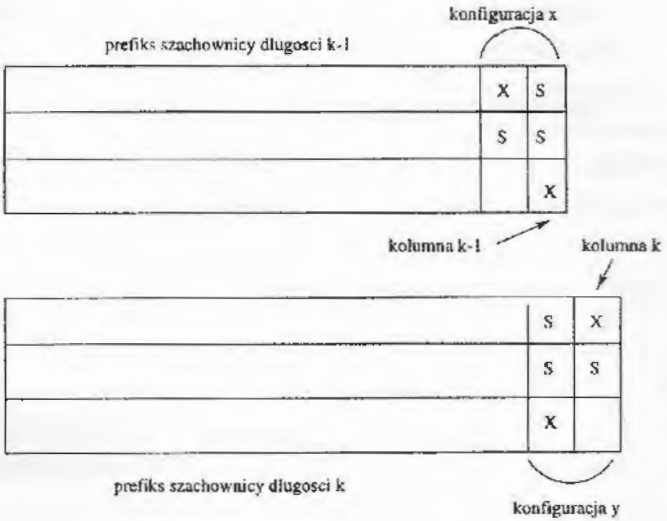
$$E = \{(x, k) \rightarrow (y, k+1): Poprzednik(x, y), (x, k), (y, k+1) \in V\}.$$

Oznaczmy przez *Waga*(x) liczbę skoczków w drugiej kolumnie konfiguracji x, a za wagę wierzchołka (x, k) przyjmijmy wagę x. Długość drogi w grafie G definiujemy, jako sumę wag wierzchołków znajdujących się na tej drodze. Zachodzi następujące twierdzenie, które wiąże liczby związane z grafem G z wielkościami, których wartości mamy wyznaczyć w rozwiązaniu zadania. Niech $v_0 = (x_0, 2)$.

Twierdzenie

(a) Max(x, k) *jest długością najdłuższej drogi z* v_0 *do* (x, k).

(b) Ile(x, k) *jest liczbą wszystkich najdłuższych dróg z* v_0 *do* (x, k).



Rysunek 2. Konfiguracje x, y. Zachodzi *Poprzednik*(x, y) oraz $(x, k-1) \to (y, k)$, *Waga*$(x) = 2$, *Waga*$(y) = 1$. Na polach oznaczonych przez S stoją skoczki, a pola oznaczone przez X należą do Z.

(c) M jest długością najdłuższej drogi rozpoczynającej się w v_0.

(d) L jest równe liczbie wszystkich dróg długości M rozpoczynających się w v_0.

Powyższe twierdzenie uzasadnia poprawność następującego schematu algorytmu rozwiązania:

Algorytm
begin
$Max(x_0, 2) := 0$; $Ile(x_0, 2) := 1$;
for $k := 3$ **to** $n+2$ **do**
for each $y \in K$ **do begin**
$Max(y, k) := \max\{Max(x, k-1) + Waga(y)\colon (x, k-1) \to (y, k)\}$;
$P := \{x\colon Max(x, k-1) = Max(y, k) - Waga(y), (x, k-1) \to (y, k)\}$;
$Ile(y, k) := \sum_{x \in P} Ile(x, k-1)$
end;
$M := \max\{Max(x, n+2)\colon x \in K\}$;
$P := \{x\colon Max(x, n+2) = M\}$;
$L := \sum_{x \in P} Ile(x, n+2)$;
write(M, L)
end

Rozwińmy teraz powyższy schemat algorytmu elimimując dosyć abstrakcyjne konstrukcje związane z pętlą **for each** oraz operacjami na zbiorach.

Przypomnijmy, że konfiguracje są określone na sześciu polach zajmujących dwie sąsiednie kolumny. Mogą więc być opisane wektorami o długości 6. Dokładniej, przyjmijmy, że konfiguracje są ponumerowane liczbami od 0 do 63 w taki sposób, że liczba i jest numerem konfiguracji, w której na polu t jest ustawiony skoczek, jeśli bit t w rozwinięciu binarnym liczby i jest równy 1, i tylko wtedy. W tej sytuacji konfiguracja pusta ma numer 0. Oczywiście nie wszystkie liczby z podanego zakresu są wykorzystane, gdyż nie każda konfiguracja jest poprawna. Dzięki tej numeracji, zamiast zmiennych x, y, do oznaczania konfiguracji można stosować zmienne i, j typu `integer`, będące ich kodami.

Istotną część realizacji algorytmu stanowi funkcja logiczna $EDGE(i, j, k)$, która ma wartość *true*, jeśli $(i, k-1) \to (j, k)$ jest łukiem w grafie G, a *false*, jeśli nie jest. Zauważmy, że $EDGE(i, j, 3) = false$, jeśli $i \neq 0$.

Załóżmy, że w pola szachownicy, na których nie można stawiać skoczków, są opisane tablicą *Dziura*, gdzie $Dziura(k)$ jest numerem pola w $k+2$ kolumnie należącego do zbioru Z, lub $Dziura(k) = 0$, jeśli żadne pole nie jest wycięte w tej kolumnie. (Przesunięcie kolumn o 2 jest związane z tym, że sztucznie dołożyliśmy dwie kolumny na początku szachownicy.)

Bardziej szczegółowy opis algorytmu ma teraz postać:

```
Algorytm
 begin
  Max(0, 2) := 0;  Ile(0, 2) := 1;
  for k := 3 to n + 2 do
   for j := 0 to 63 do begin
    max := 0;
    for i := 0 to 63 do
     if (Max(x, k - 1) + Waga(j) > max) and EDGE(i, j, k) then
      max := Max(x, k - 1) + Waga(j);
    Max(y, k) := max;
    ile := 0;
    for i := 0 to 63 do
     if (Max(i, k - 1) + Waga(j) = Max(j, k)) and EDGE(i, j, k) then
      ile := ile + Ile(i, k - 1);
    Ile(j, k) := ile
   end;
  M := 0;
```

```
  for i := 0 to 63 do
    if Max(i, n + 2) > M then M := Max(i, n + 2);
  L := 0;
  for i := 0 to 63 do
    if Max(i, n + 2) = M then L := L + Ile(i, n + 2);
  write(M, L)
end.
```

Użyliśmy dwóch pełnych tablic $Max(i, k)$ i $Ile(i, k)$, chociaż nie jest to konieczne, gdyż wykonując obliczenia w kolumnie k wystarczy znać wartości elementów tych tablic tylko dla kolumn k i $k-1$. Możemy w ten sposób znacznie zmniejszyć złożoność pamięciową algorytmu, chociaż nie jest to zbyt istotne przy rozmiarach danych, jakimi się zajmujemy. Pełna realizacja algorytmu jest zamieszczona w programie NIE.PAS.

Jest możliwe znaczne przyspieszenie tego programu, jeśli weźmie się pod uwagę, że część konfiguracji jest niepoprawna i można je pominąć w obliczeniach. Ponadto, w głównej iteracji są przeglądane wszystkie konfiguracje i oraz j w zakresie 0..63, chociaż wystarczy przeglądać tylko te j, które odpowiadają konfiguracjom mającym pierwszą kolumnę taką samą, jak i. Wtedy j miałoby 8 możliwych wartości. Wszystkie te usprawnienia zmniejszyłyby czytelność programu, dlatego je pomijamy.

Omówienie rozwiązań podanych przez uczniów

Pomimo prostego algorytmu, jego efektywna implementacja okazała się zbyt dużym problemem dla zawodników. Związane to było zapewne z trudnością podania precyzyjnej definicji funkcji logicznej *EDGE* (ewentualnie jej odpowiednika).

Większość uczniów używała konstrukcji podobnej do naszej konfiguracji.

Rozwiązania były iteracyjne, podobnie jak tutaj zaproponowane, lub rekurencyjne. Rozwiązania rekurencyjne są równie dobre w tym przypadku. Istotnym ich elementem było pamiętanie wyników, aby nie trzeba było obliczać tych samych wielkości wielokrotnie.

Znajdowanie wszystkich możliwych ustawień w sposób bezpośredni prowadziło do algorytmu działającego zbyt długo, ze względu na liczbę takich ustawień.

Testy

Użyto 11 testów do sprawdzenia rozwiązań. Pierwszy test, NIE0.IN był wzięty z treści zadania. Dwa następne, to małe testy z jedną i dwoma kolumnami. Testy NIE3.IN i NIE4.IN zawierają odpowiednio 6 i 10 kolumn i mają niewielką

liczbę rozwiązań. Dwa kolejne testy były tak dobrane, aby wykryć niepoprawne rozwiązania, realizujące metodę gry i metodę rekurencyjną (tutaj nie opisaliśmy tych metod). Test NIE7.IN zawiera losowo wygenerowany zbiór Z na szachownicy o 40 kolumnach.

Zadaniem testu NIE8.IN było sprawdzenie, czy w rozwiązaniu przewidziano dużą liczbę ustawień (przekraczającą zakres typu `integer`).

Dwa ostatnie testy są zbudowane na szachownicy o 100 kolumnach, w pierwszym z nich zbiór Z został wygenerowany losowo – w rozwiązaniu jest wiele maksymalnych ustawień, a w drugim – zbiór Z jest pusty, na szachownicy można więc ustawić maksymalną możliwą liczbę skoczków, 150, ale takich ustawień jest tylko 2.

7.3.5. Zadanie WYRÓWNYWANIE SŁÓW (Autor: Wojciech Rytter)

Treść zadania WYRÓWNYWANIE SŁÓW

Dane sa dwa słowa x i y oraz skończony ciąg słów $(w_1, w_2, \ldots, w_k)$.

Operacja $w \oplus w_i$ polega na połączeniu (konkatenacji) słowa w ze słowem w_i $(1 \le i \le k)$, czyli – mówiąc inaczej – na dopisaniu do słowa w, z prawej jego strony, słowa w_i.

Chcemy sprawdzić, czy słowa x i y można wyrównać za pomocą słów z danego ciągu. To znaczy, czy można po wykonaniu skończonej liczby operacji dopisywania do słów x i y odpowiednio wybranych wyrazów danego ciągu, otrzymać w obu przypadkach to samo słowo.

Przykład

Słowa abba oraz ab można wyrównać za pomocą wyrazów ciągu: baaabad aa badccaa cc. Wystarczy w tym celu do słowa abba dopisać: aa oraz badccaa, zaś do słowa ab – najpierw baaabad, potem cc, a następnie aa. W obu przypadkach otrzymamy to samo słowo: abbaaabadccaa.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego WYR.IN długość k danego ciągu słów, a następnie opisy dwóch słów x i y oraz opisy kolejnych słów ciągu: $(w_1, w_2, \ldots, w_k)$,
- sprawdza, czy słowa x i y można wyrównać za pomocą słów z danego ciągu i jeśli tak, znajduje minimalną liczbę operacji $\oplus$, jakie trzeba w tym celu wykonać,
- zapisuje tę liczbę w pliku tekstowym WYR.OUT.

Wejście

W pierwszym wierszu jest zapisana liczba całkowita dodatnia $k \leq 40$. Jest to długość ciągu słów $(w_1, w_2, \ldots, w_k)$.

W drugim i trzecim wierszu znajdują się opisy słów x i y.

W następnych k wierszach znajdują się opisy kolejnych słów ciągu $(w_1, w_2, \ldots, w_k)$ – każdy opis w osobnym wierszu.

Opis słowa składa się z liczby naturalnej, będącej długością tego słowa, oraz z samego słowa w postaci ciągu znaków. Liczba jest oddzielona od słowa pojedynczym odstępem.

Każde słowo składa się wyłącznie z małych liter alfabetu angielskiego od a do z i ma długość nie większą niż 2000.

Suma długości wszystkich danych słów jest nie większa niż 5000.

Wyjście

W pliku WYR.OUT należy zapisać:

- jedną liczbę całkowitą nieujemną, to jest minimalną liczbę operacji $\oplus$, jakie trzeba wykonać, aby wyrównać dane słowa x i y,
- albo jedno słowo NIE, jeśli słów nie można wyrównać.

Przykłady

Dla pliku WYR.IN:

```
4
4 abba
2 ab
7 baaabad
2 aa
7 badccaa
2 cc
```

w pliku WYR.OUT należy zapisać jedna liczbę:

```
5
```

Dla pliku WYR.IN:

```
4
1 a
2 ab
2 bb
2 ab
2 ba
2 aa
```

w pliku WYR.OUT należy zapisać słowo:

NIE

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

Opis rozwiązania zadania WYRÓWNYWANIE SŁÓW

Zauważmy na początku, że rozwiązanie sprawdzające wszystkie możliwości prowadzi do programu działającego dla pewnych danych bardzo długo, nie można go więc zaakceptować.

Zadanie jest dosyć proste, łatwo rozwiązuje się je przez sprowadzenie do problemu obliczenia długości najkrótszej drogi w pewnym pomocniczym grafie skierowanym G, chociaż bezpośrednia konstrukcja grafu G w programie nie jest konieczna.

Graf skierowany $G = (V, E)$ składa się ze zbioru **wierzchołków** V, którymi są sufiksy słów $w_1, w_2, \ldots, w_k$, i zbioru **łuków** E, gdzie $(u, v) \in E$, gdy

$$u \oplus v = w_i \text{ lub } w_i \oplus v = u$$

dla pewnego słowa w_i ze zbioru $w_1, w_2, \ldots, w_k$. **Cechą** łuku (u, v) jest liczba i – piszemy również $Następny(u, i) = v$.

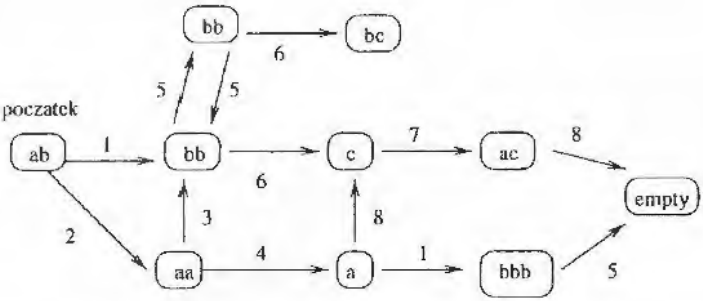
Załóżmy, że jedno z danych słów x lub y jest prefiksem drugiego, w przeciwnym bowiem przypadku mamy od razu odpowiedź NIE. Dla uproszczenia przyjmijmy, że $x = y \oplus z$. Słowo z jest **wierzchołkiem początkowym** w grafie G, oznaczamy ten wierzchołek przez v_{pocz}. Ponadto, oznaczmy przez v_{kon} wierzchołek końcowy odpowiadający słowu **pustemu** ε, dla którego zachodzi $u \oplus \varepsilon = x$.

Łatwo zauważyć, że prawdziwy jest następujący fakt.

Podstawowa własność grafu G. *Najmniejsza liczba operacji* $\oplus$ *wyrównujących teksty* x *i* y *jest równa długości (w sensie liczby łuków) najkrótszej drogi z* v_{pocz} *do* v_{kon}. *Jeśli w grafie* G *nie ma takiej drogi, to wynikiem jest słowo NIE.*

Przykład. Rozważmy słowa x = bab, y = b, oraz zbiór słów wyrównujących: w_1 = abbb, w_2 = abaa, w_3 = aabb, w_4 = aaa, w_5 = bbb, w_6 = bbc, w_7 = cac, w_8 = ac.

Wierzchołkiem początkowym jest v_{pocz} = ab, gdyż $x = y \oplus$ ab. Ta część grafu G, która nas interesuje składa się z wierzchołków osiągalnych z wierzchołka



Rysunek 1: Graf G dla przykładu z rozwiązania.

początkowego, zob. rys. 1. Najkrótsza droga z v_{pocz} do v_{kon} ma w tym przypadku długość 4 i są dwie takie najkrótsze drogi. Jedna z nich odpowiada ciągowi łuków o cechach: 1, 6, 7, 8. Odpowiada to następującemu ciągowi operacji wyrównywania:

$$y := y \oplus w_1,$$
$$x := x \oplus w_6,$$
$$y := y \oplus w_7,$$
$$x := x \oplus w_8.$$

Wyrównywanie możemy zapisać w tym przypadku również następująco:

$$x = b\ a\ b|\ b\ b\ c|\ a\ c$$
$$y = b|\ a\ b\ b\ b|\ c\ a\ c$$

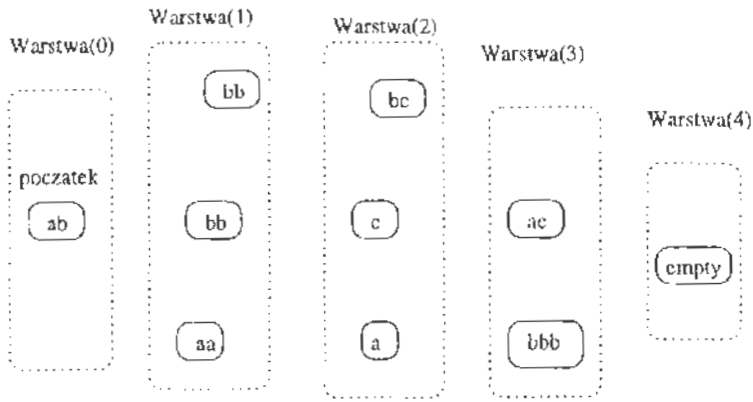
Oznaczamy przez $dist_G(v, w)$ długość najkrótszej drogi z v do w w grafie G oraz zdefiniujmy

$$Warstwa(d) = \{v :\ dist_G(v_{pocz}, v) = d\}.$$

W naszym przykładzie warstwy są zilustrowane na rys. 2.

Algorytm rozwiązania wyznacza kolejne warstwy korzystając z własności: $(d+1)$-sza warstwa składa się z wierzchołków v o postaci *Następny* (u, i), gdzie u jest dowolnym elementem poprzedniej warstwy d a v nie należy do żadnej wcześniejszej warstwy. Wynikiem końcowym działania programu jest najmniejszy numer warstwy, do której należy wierzchołek końcowy (czyli, puste słowo). Jeśli nie ma takiego wierzchołka, to program wypisujemy odpowiedź NIE.

W programie korzystamy z tablicy logicznej *Odwiedzony*, w której przechowujemy informacje czy wierzchołek należał do poprzedniej warstwy. Zakładamy, że na początku tablica ta zawiera tylko wartości *false*.



Rysunek 2: Kolejne warstwy dla przykładu z rys. 1.

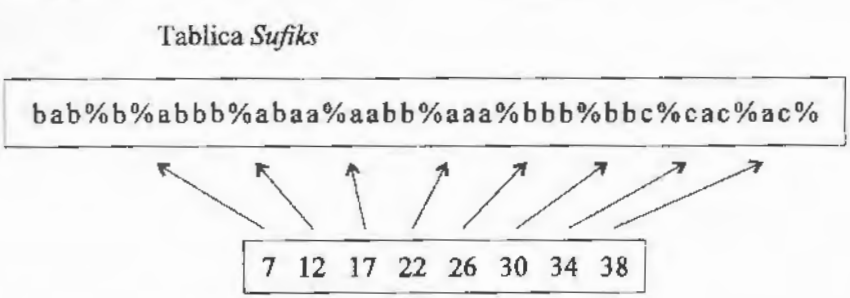
Schemat algorytmu

begin
 oblicz v_{pocz}; $Warstwa(0) := [v_{pocz}]$;
 $Odwiedzony[v_{pocz}] := true$; $d := 0$;
 while $v_{kon} \notin Warstwa(d)$ **and** $Warstwa(d) \neq \emptyset$ **do begin**
  $d := d + 1$; $Warstwa(d) := \emptyset$;
  for each $u \in Warstwa(d-1)$, $1 \leq i \leq k$ **do**
   if $Następny(u, i) \neq \emptyset$ **and not** $Odwiedzony[Następny(u, i)]$ **then begin**
    $Wstaw(Następny(u, i), Warstwa(d))$;
    $Odwiedzony[Następny(u, i)] := true$
   end
 end;
 if $v_{kon} \in Warstwa(d)$ **then** Wypisz(d) **else** Wypisz('NIE')
end.

Omówimy teraz pokrótce niektóre szczegóły realizacji tego algorytmu. Przede wszystkim warto zauważyć, że nie musimy pamiętać wszystkich warstw – wystarczy warstwy poprzednią i bieżącą. Warstwy te mogą być pamiętane w postaci stosu.

Podstawowe znaczenie ma sposób reprezentowania w programie sufiksów. Załóżmy, że w tablicy *Sufiks* są zapisane słowa x i y, a po nich – słowa $w_1, \ldots, w_k$. Słowa przedzielamy znakiem %. Oznaczmy przez x_i sufiks zaczynający się w tablicy *Sufiks* od i-tej pozycji w tej tablicy. Wtedy taki sufiks kończy się na najbliższym znaku %. Jeśli $Sufiks[i] = \%$, to x_i jest słowem pustym v_{kon}. Dla naszego przykładu mamy: x_8 = bbb, x_{14} = aa, x_{23} = aa.

Dodatkowo, w tablicy *Początek* umieszczamy numery początkowych pozycji słów w_i w tablicy *Sufiks*, zob. rys. 3.



Rysunek 3: Struktury danych dla przykładu.

Zauważmy, że kilka różnych numerów i może odpowiadać takiemu samemu słowu x_i, ale w niczym to nie przeszkadza. Moglibyśmy kilka kopii tego samego słowa zastąpić jednym, ale dzięki „powielaniu" nasza struktura danych jest bardzo prosta. W szczególności, tablica *Odwiedzony* może być indeksowana liczbami naturalnymi i dzięki temu sprawdzenie, czy *Odwiedzony*[j] = *true* można wykonać w czasie stałym.

Zauważmy, że wielkość użytej przez program pamięci jest rzędu L, gdzie L jest sumaryczną długością wszystkich danych słów.

Zadanie byłoby ciekawsze gdybyśmy zażądali, aby program wypisywał najkrótszy ciąg operacji wyrównujących.

Można również przyjąć, że słowo „najkrótszy" oznacza sumaryczną długość słów wyrównujących. Wtedy, algorytm warstwowy należałoby zmodyfikować i zamiast znajdować drogę o najmniejszej liczbie łuków szukać drogi o minimalnej sumie wag (odpowiednio zdefiuiowanych).

Wspomnijmy na końcu o problemie algorytmicznym, który jest bardzo podobny do problemu z zadania, ale jest o wiele trudniejszy. Problem **odpowiedniości słów** definiujemy następująco: Dane są dwa ciągi symboli x i y oraz dwa zbiory ciągów $w_1, w_2, ..., w_k$ i $v_1, v_2, ..., v_k$. **Zmodyfikowana operacja elementarna** polega na jednoczesnym wykonaniu:

$$x := x \oplus w_j;\ y := y \oplus v_j.$$

Pytamy się czy istnieje sekwencja zmodyfikowanych operacji elementarnych, która prowadzi do wyrównania obu ciągów, to znaczy po jej wykonaniu $x = y$. Tak zdefiniowany problem jest **nierozstrzygalny**. Oznacza to, że każdy program rozwiązujący ten problem dla pewnych danych da odpowiedź niepoprawną lub się nigdy nie zatrzyma (czyli nie da żadnej odpowiedzi).

Omówienie rozwiązań podanych przez uczniów

Zadanie okazało się stosunkowo łatwe. Wszystkie programy uczniów przetwarzały sufiksy słów bazowych. Rozwiązania różniły się szczegółami w realizacji:

1. Graf G był konstruowany albo w całości, albo tylko ta jego część, która jest przeglądana w algorytmie. Ma to wpływ na wielkość wykorzystywanej pamięci, jak również na czas obliczeń, gdyż z reguły graf G ma bardzo duże rozmiary, natomiast jego przeszukiwana część jest niewielka.
2. Funkcja *Następny*(u, i) była liczona albo bezpośrednio, albo liczono najpierw wszystkie wystąpienia słów w_i wewnątrz innych słów w_j, włącznie z wystąpieniami końcowymi, gdzie jedno słowo „wystaje" poza koniec drugiego. Można tutaj skorzystać z algorytmu Knutha-Morrisa-Pratta. Po stablicowaniu wyników, obliczanie *Następny* odbywa się w czasie stałym. Dla danych o małych rozmiarach (na przykład takich, jak w treści zadania) wydaje się, że metoda naiwna jest lepsza, szczególnie ze względu na prostotę.

Testy

Programy sprawdzano na 14 testach, z których 8 służyło sprawdzeniu poprawności rozwiązań, a 6 – testowaniu ich efektywności. Test WYR0.IN jest przykładem z tresci zadania. W dwóch następnych testach, WYR1.IN i WYR2.IN, słowa x i y są identyczne. W teście WYR3.IN nie jest spełniony warunek konieczny dla istnienia rozwiązania – żadne z danych słów nie jest prefiksem drugiego.

Dla danych z WYR4.IN nie istnieje rozwiązanie skończone; mają one postać: x = cc, y = ccab, w_1 = abcd, w_2 = abacd, w_3 = acdab.

Dla danych w testach WYR5.IN i WYR6.IN istnieją zarówno skończone, jak i nieskończone rozwiązania – w tym drugim teście jest zmieniona jedynie kolejność słów służących do wydłużania. Te testy miały na celu wykrycie rozwiązań, w których poszukiwania zapętlają się.

Test WYR7.IN jest jeszcze jednym testem poprawościowym.

Sześć ostatnich testów służyło porównaiu efektywności rozwiązań. W trzech pierwszych z nich, odpowiadajace grafy są dość gęste, ale mają względnie krótkie drogi, w trzech ostatnich – grafy są rzadkie, ale drogi odpowiadające rozwiązaniom są dość długie.

8. TEKSTY ZADAŃ Z OLIMPIADY MIĘDZYNARODOWEJ

8.1. Zadanie GRA

Rozważmy następującą grę dwuosobową. Plansza gry jest ciągiem dodatnich liczb całkowitych. Dwaj gracze wykonują ruchy na przemian. Ruch gracza polega na wybraniu liczby z lewego lub z prawego końca ciągu. Wybrana liczba jest usuwana z planszy. Gra kończy się po usunięciu wszystkich liczb. Pierwszy gracz wygrywa, gdy suma wybranych przez niego liczb jest nie mniejsza, niż suma liczb wybranych przez drugiego gracza. Drugi gracz gra najlepiej, jak tylko można. Grę rozpoczyna pierwszy gracz. Jeśli plansza zawiera początkowo parzystą liczbę elementów, to pierwszy gracz ma strategię wygrywającą.

Napisz program, który realizuje strategię prowadzącą do zwycięstwa pierwszego gracza. Reakcje drugiego gracza realizuje gotowy i dany program komputerowy. Obaj gracze komunikują się ze sobą za pomocą trzech procedur z modułu Play, z których możesz korzystać. Są to: StartGame, MyMove i YourMove (PoczątekGry, MójRuch, TwójRuch). Pierwszy gracz powinien rozpocząć grę wykonując bezparametrową procedurę StartGame. Jeśli pierwszy gracz wybiera liczbę z lewego końca, to wykonuje procedurę MyMove('L'). Podobnie, wykonanie instrukcji MyMove('R') powoduje wysłanie drugiemu graczowi komunikatu wskazującego, że pierwszy gracz wybrał liczbę z prawego końca. Drugi gracz, tzn. komputer, wykonuje ruch natychmiast i pierwszy gracz może dowiedzieć się o jego ruchu wykonując instrukcję YourMove(C), gdzie C jest typu znakowego (w języku C/C++ napiszesz to jako YourMove(&C)). Wartością C jest 'L' albo 'R', zależnie od tego czy został wybrany lewy, czy prawy koniec.

Wejście

Pierwszy wiersz w pliku INPUT.TXT zawiera rozmiar N początkowej plauszy. Liczba N jest parzysta i $2 \leq N \leq 100$. Pozostałych N wierszy zawiera początkową planszę zapisaną w kolejności od lewej do prawej; każda liczba występuje w osobnym wierszu. Każda z tych liczb jest mniejsza lub równa 200.

Wyjście

Po zakończeniu gry twój program powinien zapisać końcowy wynik gry w pliku OUTPUT.TXT. Plik ten ma zawierać w pierwszym i jedynym wierszu dwie liczby. Pierwsza z nich ma być suma liczb wybranych przez pierwszego gracza, zaś druga ma być suma liczb wybranych przez drugiego gracza. Twój program powinien grać w tę grę, a dane wyjściowe powinny odpowiadać jej przebiegowi.

Przykład danych wejściowych i wyjściowych

Poniżej przedstawiono plik wejściowy z opisem planszy początkowej i jeden z możliwych plików wyjściowych.

INPUT.TXT

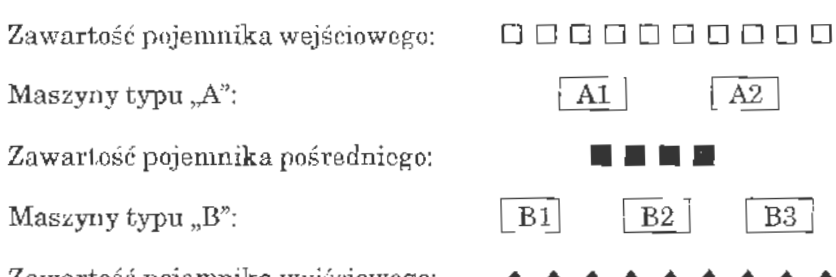
```
6
4
7
2
9
5
2
```

OUTPUT.TXT

```
15 14
```

8.2. Zadanie PRODUKCJA

W fabryce działa linia produkcyjna. Na każdym z produkowanych przedmiotów wykonuje się kolejno dwie operacje: najpierw operację „A", potem operację „B". W tej linii produkcyjnej jest pewna liczba maszyn, które mogą wykonywać te operacje. Rysunek 1 pokazuje organizację tej linii, działającej



Rysunek 1.

w następujący sposób. Maszyna typu „A" bierze przedmiot z pojemnika wejściowego, wykonuje jedną operację „A" i składa przedmiot w pojemniku pośrednim. Maszyna typu „B" bierze przedmiot z pojemnika pośredniego, wykonuje jedną operację „B" i składa przedmiot w pojemniku wyjściowym.

Wszystkie maszyny mogą pracować równolegle i niezależnie od siebie, a ładowność każdego pojemnika jest nieograniczona. Maszyny mają różne charakterystyki wyrażające się tym, że każda maszyna ma swój czas wykonania operacji.

Wyznacz najkrótszy czas, w którym można wykonać operacje „A" na wszystkich N przedmiotach, zakładając, że te przedmioty są dostępne w chwili 0. (Podzadanie A). Wyznacz także minimalny czas potrzebny do wykonania obydwu operacji na N przedmiotach (Podzadanie B).

Wejście

Plik INPUT.TXT zawiera w pięciu wierszach dodatnie liczby całkowite. Pierwszy wiersz zawiera liczbę N, czyli liczbę przedmiotów ($1 \le N \le 1000$). W drugim wierszu jest podaua liczba $M1$ maszyn typu „A" ($1 \le M1 \le 30$). W trzecim wierszu jest $M1$ liczb określających czasy wykonania operacji przez wszystkie maszyny typu „A". Czwarty i piąty wiersz zawierają odpowiednio: liczbę $M2$ maszyn typu „B" ($1 \le M2 \le 30$) i czasy wykonania operacji przez wszystkie maszyny typu „B". Podane czasy wykonania operacji, mierzone w pewnych jednostkach, uwzględniają czasy niezbędne do pobrania przedmiotu z pojemnika przed obróbką i włożenia go do pojemnika po obróbce. Każdy taki czas wykonania operacji wynosi co najmniej 1 i co najwyżej 20.

Wyjście

Twój program powinien zapisać dwa wiersze w pliku OUTPUT.TXT. Pierwszy wiersz powinien zawierać jedną liczbę całkowitą dodatnią: rozwiązanie podzadania A. Drugi wiersz powinien zawierać rozwiązanie podzadania B.

Przykład danych wejściowych i wyjściowych

Poniżej przedstawiono możliwy plik wejściowy i odpowiadający mu plik wyjściowy.

INPUT.TXT

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

OUTPUT.TXT

3
5

8.3. Zadanie SIEĆ SZKÓŁ

Pewna liczba szkół jest przyłączona do sieci komputerowej. Między tymi szkołami została zawarta pewna liczba umów: każda szkoła prowadzi listę szkół (adresatów), którym przesyła oprogramowanie. Zauważ, że jeśli B jest na liście adresatów szkoły A, to A nie musi wystąpić na liście adresatów szkoły B.

Napisz program, obliczający minimalną liczbę szkół, które muszą otrzymać kopie nowego oprogramowania, tak aby to oprogramowanie dotarło do wszystkich szkół w sieci zgodnie z umowami (Podzadanie A). Następnym podzadaniem będzie zapewnienie tego, żeby wysłanie nowego oprogramowania do dowolnie wybranej jednej szkoły powodowało przesłanie go do wszystkich szkół w sieci. Aby to osiągnąć, być może trzeba uzupełnić listy adresatów o nowe elementy. Wyznacz najmniejszą liczbę uzupełnień, które trzeba wykonać tak, aby niezależnie od wyboru szkoły, do której pośle się nowe oprogramowanie, dotarło ono do wszystkich innych szkół (Podzadanie B). Jedno uzupełnienie oznacza dołączenie jednego nowego elementu do listy adresatów jednej szkoły.

Wejście

Pierwszy wiersz pliku INPUT.TXT zawiera jedna liczbę całkowitą N: liczbę szkół w sieci ($2 < N \leq 100$). Szkoły są pooznaczane N początkowymi dodatnimi liczbami całkowitymi. Każdy z następnych N wierszy zawiera jedną listę adresatów. Wiersz $i + 1$ zawiera oznaczenia adresatów i-tej szkoły. Każda lista kończy się zerem. Samo 0 w wierszu oznacza pusta listę.

Wyjście

Twój program powinien zapisać dwa wiersze w pliku OUTPUT.TXT. Pierwszy wiersz powinien zawierać jedną dodatnią liczbę całkowitą: rozwiązanie podzadania A. Drugi wiersz powinien zawierać rozwiązanie podzadania B.

Przykład danych wejściowych i wyjściowych

Poniżej przedstawiono możliwy plik wejściowy i odpowiadający mu plik wyjściowy.

INPUT.TXT

5
2 4 3 0

```
4 5 0
0
0
1 0
```

OUTPUT.TXT

```
1
2
```

8.4. Zadanie NAJDŁUŻSZY PREFIKS

Struktura pewnych substancji biologicznych jest reprezentowana za pomocą ciągu ich elementarnych składników, które oznacza się wielkimi literami. Biologowie są zainteresowani dzieleniem długiego ciągu na pewne krótsze kawałki, które będziemy nazywać składowymi. Powiemy, że ciąg S można utworzyć ze składowych z danego zbioru P, jeśli istnieje n składowych $p_1, ..., p_n$ z P takich, że konkatenacja (złożenie) tych składowych $p_1...p_n$ równa się S. Przez konkatenację składowych $p_1, ..., p_n$ rozumiemy połączenie ich razem w tym porządku bez spacji. Ta sama składowa może występować w takiej konkatenacji więcej niż jeden raz i niekoniecznie wszystkie składowe muszą się w niej pojawić. Na przykład ciąg ABABACABAAB można utworzyć ze zbioru składowych {A, AB, BA, CA, BBC}.

Prefiksem ciągu S, o długości K, jest pierwszych K znaków w tym ciągu. Napisz program, który otrzymuje na wejściu pewien zbiór składowych P i ciąg składników elementarnych T. Twój program musi obliczyć długość najdłuższego prefiksu, który można utworzyć ze składowych z P.

Wejście

Dane wejściowe znajdują się w dwóch plikach. Plik INPUT.TXT opisuje zbiór składowych P, podczas gdy plik DATA.TXT zawiera ciąg T, który należy zbadać. Pierwszy wiersz pliku INPUT.TXT zawiera liczbę naturalna N, czyli liczbę składowych w zbiorze P ($1 \le N \le 100$). Każda składowa jest podana w dwóch kolejnych wierszach. Pierwszy wiersz zawiera jej długość L ($1 \le L \le 20$). W drugim wierszu znajduje się napis (string) o długości L, złożony z wielkich liter (od 'A' do 'Z'). Wszystkie te składowe (w liczbie N) różnią się od siebie nawzajem.

Na pierwszym miejscu w każdym wierszu (prócz ostatniego) w pliku DATA.TXT znajduje się jedna wielka litera. Plik ten jest zakończony wierszem zawierającym pojedynczą kropkę ('.'). Długość ciągu T wynosi co najmniej 1 i co najwyżej 500 000.

Wyjście

W pierwszym wierszu pliku OUTPUT.TXT zapisz długość najdłuższego prefiksu ciągu T, który można utworzyć ze zbioru P.

Przykład danych wejściowych i wyjściowych

Poniżej przedstawiono dwa pliki wejściowe i odpowiadający im plik wyjściowy.

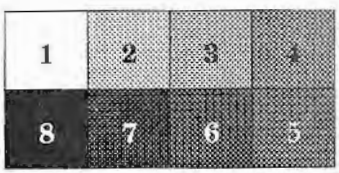
INPUT.TXT	DATA.TXT
5	A
1	B
A	A
2	B
AB	A
3	C
BBC	A
2	B
CA	A
2	A
BA	B
	C
	B

OUTPUT.TXT

11

8.5. Zadanie MAGICZNE KWADRATY RUBIKA

Po sukcesie swojej kostki Rubik wynalazł jej wersję planarną, nazywaną magicznymi kwadratami Rubika. Jest to plansza złożona z 8 kwadratów o takich samych rozmiarach (zob. rys. 1).



Rysunek 1. Konfiguracja początkowa

W tym zadaniu rozważamy wariant tej łamigłówki, w której każdy kwadrat ma inny kolor. Kolory są pooznaczane od 1 do 8 (zob. rys. 1). Układ kwadratów na planszy jest zadany przez ciąg kolorów tych kwadratów otrzymany przez odczytanie ich kolejno od lewego górnego rogu zgodnie z kierunkiem ruchu wskazówek zegara. Na przykład układ z rys. 1 jest zadany przez ciąg (1, 2, 3, 4, 5, 6, 7, 8) – jest to układ początkowy.

Na planszy można wykonywać 3 podstawowe operacje oznaczone literami 'A', 'B' i 'C':

'A': zamiana górnego wiersza z dolnym,

'B': pojedyncze przesunięcie cykliczne prostokąta w prawo,

'C': pojedynczy obrót środkowych 4 kwadratów zgodnie z kierunkiem ruchu wskazówek zegara.

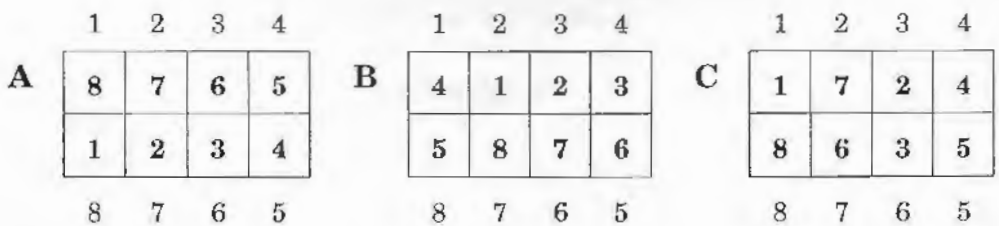
Wszystkie układy kwadratów na planszy można osiągnąć za pomocą tych 3 operacji podstawowych.

Wyniki zastosowania operacji podstawowych są przedstawione na rysunku 2. Liczby na zewnątrz kwadratów oznaczają ich pozycje. Jeśli kwadrat na pozycji p zawiera liczbę i, oznacza to, że po wykonaniu operacji kwadrat z pozycji i znalazł się na pozycji p .

Napisz program, który wyznacza ciąg operacji podstawowych przekształcających układ początkowy z rys. 1 w podany układ końcowy (Podzadanie A). Dodatkowe dwa punkty można zdobyć za rozwiązanie, w którym długość ciągu operacji podstawowych nie przekracza 300 (Podzadanie B).

Wejście

Plik INPUT.TXT zawiera w pierwszym wierszu 8 dodatnich liczb całkowitych, stanowiących opis końcowego układu kwadratów.



Rysunek 2. Operacje podstawowe

Wyjście

W pierwszym wierszu pliku OUTPUT.TXT twój program musi zapisać długość L ciągu operacji podstawowych. W kolejnych L wierszach należy zapisać ciąg identyfikatorów operacji podstawowych; po jednej literze na pierwszym miejscu każdego wiersza.

Program pomocniczy

MTOOL.EXE jest programem, znajdującym się w katalogu tego zadania, który pozwala bawić się kwadratami Rubika. Wykonując MTOOL INPUT.TXT OUPUT.TXT możesz eksperymentować z układem końcowym i ciągiem operacji podstawowych.

Przykład danych wejściowych i wyjściowych

Poniżej przedstawiono przykładowy pliki wejściowe i odpowiadający mu plik wyjściowy.

INPUT.TXT

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

OUTPUT.TXT

```
7
B
C
A
B
C
C
B
```

8.6. Zadanie SORTOWANIE CIĄGU TRÓJWARTOŚCIOWEGO

Sortowanie jest jednym z najczęściej wykonywanych zadań obliczeniowych. Rozważmy szczególny problem sortowania rekordów, których klucze mają co najwyżej 3 różne wartości. Dzieje się tak na przykład, gdy sortujemy medalistów na zawodach w kolejności otrzymanych medali, tzn. złoci medaliści mają się pojawić jako pierwsi, po nich srebrni, a na końcu brązowi.

W tym zadaniu możliwymi wartościami kluczy są liczby całkowite 1, 2 i 3. Klucze należy posortować niemalejąco. Sortowanie ma być wykonane za pomocą ciągu operacji zamiany. Operacja zamiany, określona przez dwa numery pozycji p i q, polega na zamianie elementów z pozycji p i q.

Dany jest ciąg wartości kluczy. Napisz program, który oblicza najmniejszą liczbę takich zamian, które są konieczne do posortowania tego ciągu (Podzadanie A). Ponadto wyznacz konkretny ciąg zamian, który takie posortowanie zrealizuje (Podzadanie B).

Wejście

Pierwszy wiersz pliku INPUT.TXT zawiera liczbę rekordów N ($1 \leq N \leq 1000$). Każdy z następnych N wierszy zawiera pojedynczą wartość klucza.

Wyjście

W pierwszym wierszu pliku OUTPUT.TXT zapisz minimalną liczbę L zamian koniecznych do posortowania ciągu wejściowego (Podzadanie A). Następnych L wierszy ma zawierać realizujący takie sortowanie ciąg zamian, w kolejności ich wykonania. Każdy z tych wierszy powinien zawierać jedną zamianę opisaną za pomocą dwóch liczb p i q, które są numerami pozycji zamienianych elementów (Podzadanie B). Pozycje są numerowane liczbami całkowitymi od 1 do N.

Przykład danych wejściowych i wyjściowych

Poniżej przedstawiono pewien plik wejściowy i pewien odpowiadający mu plik wyjściowy.

INPUT.TXT

```
9
2
2
1
3
3
3
2
3
1
```

OUTPUT.TXT

```
4
1 3
4 7
9 2
5 9
```

LITERATURA

Poniżej zamieszczamy listę książek w języku polskim, które są poświęcone algorytmom i ich kompletnym realizacjom. Do wielu z tych książek odwołujemy się w rozwiązaniach zadań, mogą być również przydatne w przygotowywaniu się do zawodów następnych olimpiad.

1. *I Olimpiada Informatyczna 1993/1994*, Warszawa, Wrocław 1994.
2. *II Olimpiada Informatyczna 1994/19945*, Warszawa, Wrocław 1994.
3. Aho, A.V., Hopcroft, J.E., Ullman, J.D., *Projektowanie i analiza algorytmów komputerowych*, PWN, Warszawa 1983.
4. Banachowski, L., Kreczmar, A., *Elementy analizy algorytmów*, WNT, Warszawa 1982.
5. Banachowski, L., Kreczmar, A., Rytter, W., *Analiza algorytmów i struktur danych*, WNT, Warszawa 1987.
6. Banachowski, L., Diks, K., Rytter, W., *Algorytmy i struktury danych*, WNT, Warszawa 1996.
7. Bentley, J., *Perełki oprogramowania*, WNT, Warszawa 1992.
8. Bronsztejn, I.N., Siemiendiajew, K.A., *Matematyka. Poradnik encyklopedyczny*, Wydawnictwo Naukowe PWN, Warszawa 1996.
9. Corman, T.H., Leiserson, C.E., Rivest, R.L., *Wprowadzenie do algorytmiki*, WNT, Warszawa 1996.
10. *Elementy Informatyki. Pakiet oprogramowania edukacyjnego*, Instytut Informatyki Uniwersytetu Wrocławskiego, OFEK, Wrocław, Poznań 1993.
11. *Elementy informatyki: Podręcznik (vol. 1), Rozwiązania zadań (vol. 2), Poradnik metodyczny dla nauczyciela (vol. 3)*, pod redakcją M.M. Sysły, Wydawnictwo Naukowe PWN, Warszawa 1996.
12. Graham, R., Knuth, D.E., Patashnik, O., *Matematyka konkretna*, Wydawnictwo Naukowe PWN, Warszawa 1996.
13. Harel, D., *Algorytmika – Rzecz o istocie informatyki*, WNT, Warszawa 1992.
14. Lipski, W., *Kombinatoryka dla programistów*, WNT, Warszawa 1989.

15. Reingold, E.M., Nievergelt, J., Deo, N., *Algorytmy kombinatoryczne*, WNT, Warszawa 1985.
16. Ross, K.A., Wright, C.R.B., *Matematyka dyskretna*, Wydawnictwo Naukowe PWN, Warszawa 1996.
17. Sysło, M.M., Deo, N., Kowalik, J.S., *Algorytmy optymalizacji dyskretnej z programami w języku Poscal*, Wydawnictwo Naukowe PWN, Warszawa 1993.
18. Wilson, R.J., *Wprawadzenie do teorii grafów*, PWN, Warszawa 1985.
19. Wirth, N., *Algorytmy + struktry danych = programy*, WNT, Warszawa 1989.