MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

I OLIMPIADA INFORMATYCZNA

1993/1994

WARSZAWA, WROCŁAW – 1994

Autorzy:

Rozdz. 1–5: Komitet Główny Olimpiady
Rozdz. 6: dr Andrzej Walat i prof. dr hab. inż. Stanisław Waligórski
Rozdz. 7: Autorzy zadań: dr Piotr Chrząstowski-Wachtel, mgr inż. Wojciech Complak, prof. dr hab. Wojciech Rytter, prof. dr hab. Maciej M. Sysło, dr Andrzej Walat
Rozdz. 8: prof. dr hab. inż. Stanisław Waligórski
Rozdz. 9: dr Andrzej Walat i prof. dr hab. inż. Stanisław Waligórski – przekłady tekstów zadań

Opracowanie i redakcja:

Prof. dr hab. Maciej M. Sysło

Skład komputerowy:

Magdalena Kraszewska

© Copyright by Komitet Główny Olimpiady Informatycznej

Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
ul. Raszyńska 8/10, 02-026 Warszawa

Nakład 2000 egz.

SPIS TREŚCI

1. WSTĘP

W roku szkolnym 1993/1994 odbyła się I Olimpiada Informatyczna w Polsce.

Zawody informatyczne dla młodzieży mają u nas w kraju już dość długą historię. Początkowo były to przede wszystkim zawody o ograniczonym, zwykle lokalnym zasięgu, organizowane przez entuzjastów-nauczycieli i niektóre organizacje wspierajace szkoły. W ostatnim okresie, przez dwa kolejne lata 1991–1992 był organizowany Krajowy Konkurs Informatyczny, którego zasięg był ogólnokrajowy, ale z pewnością informacje o nim docierały do niewielu szkół. Spośród laureatów tego Konkursu tworzono zespół reprezentujący Polskę na kolejnych Olimpiadach Międzynarodowych.

W grudniu 1993 roku, po spełnieniu wymogów zarządzenia Ministra Edukacji Narodowej w sprawie organizacji olimpiad przedmiotowych, Instytut Informatyki Uniwersytetu Wrocławskiego powołał Olimpiadę Informatyczną oraz jej Komitet Główny złożony z osób, uprzednio zaangażowanych w organizowanie Konkursu.

Oddajemy do rąk Czytelników szczegółową relację z przebiegu I Olimpiady Informatycznej oraz z zawodów międzynarodowych, w których brali udział zdobywcy czterech pierwszych miejsc w olimpiadzie krajowej.

Ponieważ Olimpiada Informatyczna była organizowana u nas w kraju po raz pierwszy, zawarliśmy w tym opracowaniu również oficjalne dokumenty Komitetu Głównego: Akt Powołania Olimpiady Informatycznej, jej Regulamin i Zasady organizacji zawodów w 1993/1994 roku. Zwłaszcza te dwa ostatnie dokumenty zawierają wiele informacji ważnych dla uczestników następnych 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. W rozdziale poświęconym metodom sprawdzania i oceniania rozwiązań zwracamy ponadto uwagę, że od zawodników wymaga się nie tylko umiejętności programowania, doboru algorytmów i biegłości w posługiwaniu się komputerem, ale również spełniania for-

malnych wymagań dotyczących postaci rozwiązań, w tym wyników oraz danych wejściowych dla tworzonych programów.

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 rozwiązuje zadanie i robi to możliwie sprawnie 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 Czytelników do dalszego pogłębiania swojej wiedzy i umiejętności oraz sięgnięcia po podręczniki poświęcone algorytmice, algorytmom i strukturom danych i metodom obliczeniowym z różnych dziedzin.

W 1994 roku wprowadzono po raz pierwszy komputerowo wspomagane sprawdzanie rozwiązań zadań olimpiad informatycznych: krajowej i międzynarodowej. Dotychczas, prace były sprawdzane na komputerze indywidualnie i osoba sprawdzająca oceniała działanie programu oraz produkowane przez niego wyniki. W rozdziale 8 wyjaśniamy bliżej na czym polega rola komputera w sprawdzaniu zadań i jakie problemy pojawiają się w związku ze stosowaniem tej nowej metody postępowania. Wprowadzając komputerowe wspomaganie sprawdzania i oceniania prac I Olimpiady Informatycznej staraliśmy się uczynić to przejście możliwie łagodnym: nasz program sprawdzający uwzględnia rozwiązania częściowe, toleruje też niektóre usterki i drobniejsze błędy w rozwiązaniach. Komitet Naukowy VI Międzynarodowej Olimpiady nie postępował tak ostrożnie i zastosował zasadę oceniania wyników testów programów „wszystko albo nic". Nie tolerowano w rozwiązaniach praktycznie żadnych odstępstw od treści zadań i drobne usterki dyskwalifikowały tak samo, jak poważniejsze błędy. Źródeł tak rygorystycznej metody sprawdzania rozwiązań zadań w Olimpiadzie Międzynarodowej można upatrywać częściowo w liczbie zawodników biorących udział w tych zawodach oraz zbyt krótkim czasie na dokładniejsze, ręczne sprawdzenie rozwiązań.

Staraliśmy się, by to opracowanie trafiło do rąk Czytelników jeszcze przed rozesłaniem pierwszych zadań II 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.

2. AKT POWOŁANIA OLIMPIADY INFORMATYCZNEJ

Na podstawie zarządzenia nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku w sprawie organizacji konkursów i olimpiad przedmiotowych Instytut Informatyki Uniwersytetu Wrocławskiego powołuje Olimpiadę Informatyczną.

Zasady jej prowadzenia reguluje załączony Regulamin Olimpiady Informatycznej*.

Instytut Informatyki Uniwersytetu Wrocławskiego powołuje Komitet Główny Olimpiady Informatycznej w składzie:

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;
prof. dr hab. Maciej M. Sysło	– Uniwersytet Wrocławski;
prof. dr hab. inż. Stanisław Waligórski	– Uniwersytet Warszawski;
dr Piotr Chrząstowski-Wachtel	– Uniwersytet Warszawski;
dr Andrzej Walat	– Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie;
dr Bolesław Wojdyło	– Uniwersytet Mikołaja Kopernika w Toruniu;
mgr Jerzy Dałek	– Ministerstwo Edukacji Narodowej;
mgr Krzysztof J. Święcicki	– Ministerstwo Edukacji Narodowej;
Tadeusz Kuran	– Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie;

* Pełny tekst Regulaminu Olimpiady Informatycznej jest zamieszczony w rozdziale 3.

mgr Krystyna Kominek – sekretarz Komitetu Głównego, II Liceum Ogólnokształcące im. Stefana Batorego w Warszawie

Siedzibą Olimpiady jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów, 02-026 Warszawa, ul. Raszyńska 8/10, współdziałający z Instytutem Informatyki Uniwersytetu Wrocławskiego na zasadach określonych w Deklaracji Ośrodka z dnia 8.12.1993 r.

3. 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 tym regulaminie, związane z utworzeniem Komitetów Okręgowych Olimpiady, których głównym zadaniem jest organizacja zawodów II stopnia. W roku szkolnym 1994/95 utworzone zostaną trzy Komitety Okręgowe: w Toruniu, Warszawie i we Wrocławiu. Zmodyfikowany regulamin zostanie dostarczony do kuratoriów i szkół wraz z zadaniami zawodów I stopnia II Olimpiady Informatycznej.

§ 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. Liczbę i treść zadań na każdy stopień zawodów ustala Komitet Główny, wybierając je drogą głosowania spośród zgłoszonych projektów, odsyłając do recenzji i przyjmując ostateczny tekst, przedstawiany wspólnie przez autora i recenzenta, na kolejnym posiedzeniu Komitetu.
5. 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.
6. Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz nadesłaniu rozwiązań pod adresem odpowiedniego Komitetu Olimpiady Informatycznej w podanym terminie.
7. Liczbę uczestników zakwalifikowanych do zawodów II i III stopnia ustala Komitet Główny i podaje ją w „Zasadach organizacji zawodów" na dany rok szkolny.
8. 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.
9. Komitet Główny Olimpiady kwalifikuje do zawodów II i III stopnia odpowiednią liczbę uczestników, których rozwiązania zadań stopnia niższego ocenione zostaną najwyżej.
10. Zawody II i III stopnia polegają na samodzielnym rozwiązaniu przez uczestników Olimpiady zakwalifikowanych do tych zawodów zadań przygotowanych dla danego stopnia w ciągu dwóch sesji przeprowadzanych w różnych dniach w warunkach kontrolowanej samodzielności.
11. 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, w miejsce ustępujących, w liczbie nie większej niż 1/3 liczby wszystkich członków Komitetu. W przypadku rezygnacji większej liczby członków organizator powoła nowy skład Komitetu.
5. 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 odpowiedzialne jest 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.
6. Decyzje Komitetu zapadają zwykłą większością głosów przy obecności przynajmniej połowy członków. W przypadku równej liczby głosów decyduje głos przewodniczącego obrad.
7. Posiedzenia Komitetu, na których ustala się pisemne tematy Olimpiady są tajne. Przewodniczący obrad może zarządzić tajność obrad także w innych uzasadnionych przypadkach.
8. Decyzje Komitetu we wszystkich sprawach dotyczących zadań i uczestników są ostateczne.

9. Komitet dysponuje funduszem Olimpiady za pośrednictwem kierownika organizacyjnego Olimpiady.
10. 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.
11. Pracami Komitetu kieruje przewodniczący, a w jego zastępstwie lub z jego upoważnienia wiceprzewodniczący.
12. 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.
13. 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. 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 poprzedzone jest 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.

§ 6. 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.

§ 7. 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.

§ 8. 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 będzie powoływał Komitety Okręgowe Olimpiady, w miarę powiększania się liczby uczestników zawodów i powstawania odpowiednich warunków organizacyjnych.
4. Komitet Główny zatwierdza sprawozdanie z przeprowadzonej Olimpiady w ciągu 2 miesięcy po jej zakończeniu i przedstawia je organizatorowi i Ministerstwu Edukacji Narodowej.
5. 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.

4. ZASADY ORGANIZACJI ZAWODÓW W ROKU SZKOLNYM 1993/94

Podstawowym aktem prawnym dotyczącym Olimpiady jest jej Regulamin, którego pełny tekst znajduje się w kuratoriach oświaty. Niniejsze „Zasady organizacji zawodów" są uzupełnieniem Regulaminu Olimpiady Informatycznej* zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 1993/94.

§ 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 r.

§ 2. 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 dla młodzieży (z wyjątkiem szkół policealnych i wyższych uczelni).
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, Lisp, C, C++, Prolog). Rozwiązania w asemblerze nie będą brane pod uwagę.
5. Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu 3 zadań i nadesłaniu rozwiązań w podanym terminie.

* Pełny tekst Regulaminu Olimpiady Informatycznej jest zamieszczony w rozdziale 3.

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 60 uczestników, którzy uzyskają najwięcej punktów za rozwiązania zadań I stopnia; do zawodów III stopnia – 30 uczestników, którzy uzyskają najwięcej punktów za rozwiązania zadań II stopnia. 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 i nadesłaniu rozwiązań pocztą, **przesyłką poleconą**, pod adresem:

Olimpiada Informatyczna
Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
ul. Raszyńska 8/10, 02-026 Warszawa

w nieprzekraczalnym terminie nadania do poniedziałku dnia 24.01.1994 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki.
Rozwiązanie wszystkich zadań nie jest warunkiem udziału w Olimpiadzie.
2. Rozwiązanie każdego zadania składa się:
   a) z jednego programu na dyskietce,
   b) z wydrukowanego tekstu tego programu w postaci źródłowej,
   c) z opisu algorytmu rozwiązania zadania wraz z uzasadnieniem jego poprawności.
3. 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 urodzenia,
   — dokładny adres zamieszkania i ewentualnie numer telefonu,
   — nazwę, adres i numer telefonu szkoły, do której uczęszcza,

   — opis konfiguracji komputera, na którym rozwiązał zadania,
   — nazwę i numer wersji użytego języka programowania.
4. Uczestnik przysyła jedną dyskietkę, oznaczoną jego imieniem i nazwiskiem, nadającą się do uruchomienia na komputerze IBM PC i zawierającą:
   — wszystkie programy w postaci źródłowej i skompilowanej,
   — spis zawartości dyskietki w pliku nazwanym SPIS.TRC.

   Imię i nazwisko uczestnika powinno być podane w komentarzu na początku każdego programu.
5. Nazwy plików z programami w postaci źródłowej powinny mieć jako rozszerzenie co najwyżej trzyliterowy skrót nazwy tego języka programowania, w szczególności

Pascal	PAS
Basic	BAS
Logo	LOG
Lisp	LIS
C	C
C++	C
PROLOG	PRO .

6. Opcje kompilatora powinny być częścią tekstu programu. Zaleca się stosowanie opcji standardowych.
7. Plik wejściowy każdego zadania jest niepustym plikiem tekstowym, w którym może być wiele zestawów danych wejściowych tego zadania. W takim przypadku kolejne zestawy danych wejściowych są oddzielane jednym pustym wierszem. Po ostatnim zestawie następuje znak końca pliku.
8. Plik wyjściowy każdego zadania ma być niepustym plikiem tekstowym, zawierającym ciąg odpowiedzi dla kolejnych zestawów danych wejściowych z pliku wejściowego – jedną odpowiedź dla każdego zestawu danych. Kolejne odpowiedzi powinny być oddzielane pustym wierszem. Po ostatniej odpowiedzi powinien być znak końca pliku.
9. Oprócz odpowiedzi określonych w sformułowaniu każdego zadania wynikiem działania programu może być – dla dowolnego z zadań – słowo KAPITULACJA oznaczające, że program nie jest w stanie znaleźć żądanego rozwiązania. Słowo to powinno być wpisane do pliku wyjściowego tak jak każda inna odpowiedź.
10. Prace zbiorowe, niesamodzielne lub nie spełniające warunków opisanych w punktach 2 – 6 nie będą brane pod uwagę.

§ 4. Uprawnienia i nagrody

1. Uczestnicy zawodów II i III stopnia 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 dojrzałości (zgodnie z zarządzeniem nr 24 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.
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 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65, poz. 385), o ile te uchwały nie stanowią inaczej.
5. Komitet Główny kwalifikuje do reprezentacji Polski na VI Międzynarodową Olimpiadę Informatyczną w 1994 roku czterech laureatów, którzy:
   - uzyskali najwyższą punktację w zawodach III stopnia,
   - są obywatelami polskimi oraz
   - nie osiągnęli 20 lat do dnia 3 lipca 1994 r. (wymóg regulaminu Międzynarodowej Olimpiady Informatycznej).
6. Zaświadczenia o uzyskanych uprawnieniach wydaje uczestnikom Komitet Główny.
7. Nauczyciel (opiekun naukowy), który przygotował laureata Olimpiady Informatycznej, otrzymuje nagrodę przyznawaną przez Komitet Główny Olimpiady.

§ 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 uczestników o ich zakwalifikowaniu bądź nie zakwalifikowaniu do kolejnego etapu, a każdego uczestnika, który przeszedł do zawodów wyższego stopnia oraz dyrektora jego szkoły także 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 trzech zadań zawodów I stopnia, a po nich – następujące sugestie dla rozwiązujących zadania:

Twoja praca będzie wyżej oceniona
- **gdy będzie kompletna i poprawna,**
- **gdy rozwiązania będą miały dobrą strukturę i będą zrozumiałe,**
- **gdy programy będą przejrzyste i czytelne,**
- **gdy wybierzesz metodę rozwiazania zadania, która w złożonych przypadkach daje wyniki w krótszym czasie niż inne metody.**

5. SPRAWOZDANIE Z PRZEBIEGU I 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 rozdz. 2).

Podstawowym aktem prawnym dotyczącym Olimpiady jest jej Regulamin (zob. rozdz. 3). Uzupełnieniem Regulaminu są „Zasady organizacji zawodów" (zob. rozdz. 4), które zawierają szczegółowe postanowienia o organizacji zawodów w roku szkolnym 1993/1994.

Celami Olimpiady są między innymi: rozszerzanie współpracy nauczycieli akademickich z nauczycielami szkół w kształceniu młodzieży uzdolnionej, stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej, stworzenie motywacji dla zainteresowania nauczycieli i uczniów nowymi metodami informatyki.

Ranga Olimpiady Informatycznej jest potwierdzona trzyletnim doświadczeniem jej organizatorów w prowadzeniu Krajowych Konkursów Informatycznych pod patronatem Ministerstwa Edukacji Narodowej.

Instytut Informatyki Uniwersytetu Wrocławskiego w Akcie Powołania Olimpiady Informatycznej (rozdz. 2) powołał Komitet Główny Olimpiady Informatycznej w składzie:

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;
Prof. dr hab. Maciej M. Sysło	– Uniwersytet Wrocławski;
Prof. dr hab. inż. Stanisław Waligórski	– Uniwersytet Warszawski;
Dr Piotr Chrząstowski-Wachtel	– Uniwersytet Warszawski;

Dr Andrzej Walat	– Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie;
Dr Bolesław Wojdyło	– Uniwersytet Mikołaja Kopernika w Toruniu;
Mgr Jerzy Dałek	– Ministerstwo Edukacji Narodowej;
Mgr Krzysztof J. Święcicki	– Ministerstwo Edukacji Narodowej;
Tadeusz Kuran	– Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie;
Mgr Krystyna Kominek	– sekretarz Komitetu Głównego, II Liceum Ogólnokształcące im. Stefana Batorego w Warszawie.

Komitet Główny Olimpiady Informatycznej wybrał następujących członków Prezydium:

przewodniczący	– prof. dr hab. inż. Stanisław Waligórski,
zastępca przewodniczącego	– prof. dr hab. Maciej M. Sysło,
sekretarz naukowy	– dr Andrzej Walat,
kierownik organizacyjny	– Tadeusz Kuran,
sekretarz	– mgr Krystyna Kominek.

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

Komitet Główny odbył łącznie z zebraniami grupy inicjatywnej Olimpiady, której skład pokrywał się ze składem Komitetu Głównego, 9 posiedzeń.

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

mgr Jakub Bojanowski	– pracownik IIUW,
mgr Piotr Filip Sawicki	– pracownik IIUW,
Marcin Kubica	– student V roku,
Marcin Engel	– student V roku,
Marek Pawlicki	– student IV roku,
Krzysztof Stencel	– student IV roku,
Marcin Madey	– student III roku,
Marcin Jurdziński	– student II roku,

Piotr Krysiuk – student II roku,
Tomasz Śmigielski – student II roku.

Olimpiada Informatyczna jest trójstopniowa. Zawody I stopnia miały charakter otwartego konkursu przeprowadzonego dla uczniów wszystkich typów szkół młodzieżowych, a zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Czołówka laureatów I Olimpiady wzięła udział w regionalnym Konkursie Informatycznym Krajów Europy Centralnej w Cluj, Rumunia, w dniach 26–31 maja 1994 r. (zob. p. 6.2 i 9.1) i reprezentowała Polskę na VI Międzynarodowej Olimpiadzie Informatycznej w dniach 3–10 lipca 1994 r. w Sztokholmie (zob. p. 6.3 i 9.2).

Zgodnie z przepisami ustawy z dnia 12 września 1990 roku o szkolnictwie wyższym (Dz.U. nr 65, poz. 385) na mocy uchwał senatów poszczególnych uczelni laureaci zawodów III stopnia, a także finaliści, mogą być zwolnieni w części lub w całości z egzaminów wstępnych do szkół wyższych.

Komitet Główny nawiązał kontakt z wyższymi uczelniami w Polsce w sprawie przyjmowania laureatów i finalistów Olimpiady Informatycznej bez egzaminów na I rok studiów. Odpowiedziało 21 uczelni, z czego: 7 zadeklarowało chęć przyjmowania bez egzaminu wszystkich laureatów i finalistów w roku akademickim 1994/5; 3 uczelnie będą preferować finalistów przy egzaminach; 6 uczelni podjęło decyzje o przyjmowaniu bez egzaminów w roku akademickim 1995/6. Cztery uczelnie odmówiły przyjmowania bez egzaminów wstępnych finalistów i laureatów Olimpiady Informatycznej, najczęściej z powodu braku zbliżonych do informatyki kierunków studiów.

Zawody I stopnia

Zawody I stopnia rozpoczęły się dnia 3 stycznia 1994 roku. Ostatecznym terminem nadsyłania prac konkursowych był dzień 24 stycznia 1994 roku.

W zawodach I stopnia Olimpiady wzięło udział 528 uczniów, którzy nadesłali łącznie 1380 rozwiązań zadań, w tym

521 rozwiązań zadania TRÓJKĄTY,
480 rozwiązań zadania SPONSOR,
334 rozwiązań zadania PIONKI – łącznie 1335 rozwiązań

oraz w kopiach zapasowych:

15 rozwiązań zadania TRÓJKĄTY,
16 rozwiązań zadania SPONSOR,
14 rozwiązań zadania PIONKI – łącznie 45 rozwiązań.

Dwóch uczniów przysłało dyskietki uszkodzone fizycznie. Za pomocą odpowiedniego programu odzyskano zapis na jednej z nich, niestety druga dyskietka

okazała się nieczytelna. Na 12 dyskietkach zauważono i usunięto różnego rodzaju programy wirusowe.

17 uczniów nadesłało prace po wyznaczonym terminie 24 stycznia 1994r., co zostało potwierdzone datą na stemplu pocztowym. Jeden z nich odwołał się od decyzji sekretariatu i okazało się, że urząd pocztowy przyjmując jego przesyłkę w dniu 24.01.1994 r. przystawił stempel z datą dnia następnego. Praca jego była sprawdzona i oceniona w terminie późniejszym. Wynik jaki osiągnął nie kwalifikowałby go jednak do zawodów II stopnia.

Komputerowe wspomaganie umożliwiło sprawdzenie rozwiązań zadań I stopnia kompletem 39 testów w ciągu kilkunastu godzin. Sprawdzanie było utrudnione występowaniem w rozwiązaniach takich niedokładności, jak nieprzestrzeganie podanych w treści zadań reguł dotyczących nazywania plików i tworzenia zestawów danych wynikowych.

Wyniki związane z zadaniem TRÓJKĄTY były najsłabsze, pomimo że nadesłano najwięcej rozwiązań tego zadania. Najlepszy rezultat 87 pkt. osiągnął 1 uczeń, a 0 pkt. otrzymało 58 uczniów, co stanowi ok. 11% wszystkich nadesłanych rozwiązań tego zadania.

Wyniki związane z zadaniem SPONSOR były lepsze – 100 pkt. otrzymało 12 uczniów, czyli 3%, chociaż 73 uczniów otrzymało 0 pkt., co stanowi ok. 15% wszystkich nadesłanych rozwiązań tego zadania.

Wyniki związane z zadaniem PIONKI były najlepsze – maksymalną liczbę 100 pkt. uzyskało 59 uczniów, czyli 18%, a 54 uczniów otrzymało 0 pkt., co stanowi ok. 16% wszystkich nadesłanych rozwiązań tego zadania.

W zawodach I stopnia Olimpiady najliczniej były reprezentowane województwa:

st. warszawskie	– 83 uczniów,	poznańskie	– 17,
katowickie	– 49,	toruńskie	– 15,
wrocławskie	– 42,	lubelskie	– 14,
krakowskie	– 35,	opolskie	– 14,
kieleckie	– 24,	rzeszowskie	– 14,
gdańskie	– 23,	szczecińskie	– 13,
łódzkie	– 18,	bielskie	– 11,
tarnowskie	– 18,	legnickie	– 10.

Nie zgłosili się do konkursu uczniowie z województw: chełmskiego, konińskiego, włocławskiego i zamojskiego.

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

Warszawa	– 78 uczniów,	Kielce	– 12,
Kraków	– 32,	Szczecin	– 12,
Wrocław	– 31,	Lublin	– 11,
Łódź	– 16,	Gdańsk	– 10,
Poznań	– 15,	Ostrowiec Św.	– 10,
Toruń	– 15,	Dębica	– 8,
Katowice	– 12,	Mielec	– 6.

Zawody II stopnia

Zawody II stopnia odbyły się w OEIiZK w Warszawie w dniach 18–20 marca 1994 r.

Do zawodów II stopnia Komitet Główny Olimpiady, na podstawie ocen rozwiązań dostarczonych przez jury Olimpiady, zakwalifikował 64 uczniów, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 188 pkt.

Najliczniej były reprezentowane szkoły z następujących miast:

Warszawy	– 15 uczniów,	Krakowa	– 3,
Torunia	– 5,	Mielca	– 3,
Gdyni	– 4,	Wrocławia	– 3.

Dnia 18 marca odbyła się sesja próbna, w czasie której uczniowie rozwiązywali nie liczące się w ogólnej klasyfikacji zadanie SUMA KWADRATÓW CYFR (w skrócie SUMA). W dwóch następnych dniach rozwiązywali zadania konkursowe: PRZEDSIĘWZIĘCIE i WYSPY NA TRÓJKĄTNEJ SIECI (w skrócie WYSPY).

Za zadanie PRZEDSIĘWZIĘCIE maksymalną liczbę 100 pkt. otrzymało 15 uczniów, tj. ok. 23%, pozostali otrzymali mniej niż 90 pkt., a 0 pkt. otrzymało 9 uczniów, tj. ok. 14%.

Za zadanie WYSPY maksymalną liczbę 100 pkt. otrzymało 3 uczniów, tj. ok. 5% i 9 uczniów miało ponad 90 pkt., a 0 pkt. otrzymało 16 uczniów, tj. ok. 25%.

Zawody III stopnia

Zawody III stopnia odbyły się w w dniach od 18–22 kwietnia 1994 roku, również w OEIiZK w Warszawie

W zawodach III stopnia wzięło udział 33 najlepszych uczestników zawodów II stopnia, którzy uzyskali nie mniej niż 97 pkt., z województw:

st. warszawskiego	– 6,	krakowskiego	– 2,
katowickiego	– 3,	poznańskiego	– 2,
rzeszowskiego	– 3,	toruńskiego	– 2,
gdańskiego	– 2,	wrocławskiego	– 2,

oraz po 1 uczniu z województw, lubelskiego, łódzkiego, nowosądeckiego, opolskiego, radomskiego, suwalskiego, szczecińskiego, tarnowskiego i zielonogórskiego.

Dnia 18 kwietnia odbyła się sesja próbna, w czasie której uczniowie rozwiązywali nie liczące się w ogólnej klasyfikacji zadanie WAHANIA AKCJI NA GIEŁDZIE (w skrócie GIEŁDA).

Pierwszego dnia konkursowego 19 kwietnia uczniowie rozwiązywali dwa zadania ANAGRAMY i NASYCANIE MAKROFANÓW (w skrócie MAKROFANY), każde za 50 pkt., w tym 47 pkt. za testy i 3 pkt. za rozwiązania uznane przez jury za godne szczególnego wyróżnienia.

Za zadanie ANAGRAMY maksymalną liczbę 47 pkt. otrzymało 4 uczniów, tj. ok. 12%, pozostali otrzymali mniej niż 40 pkt., a 0 pkt. nie otrzymał żaden z uczniów. Żadnemu z uczniów jury nie przyznało punktów uznaniowych.

Za zadanie MAKROFANY maksymalną liczbę 47 pkt. otrzymało 7 uczniów, tj. ok. 21%, 5 uczniów otrzymało ponad 40 pkt., a 0 pkt. otrzymało 3 uczniów, tj. ok. 9%. Jury przyznało 3 lub 2 punkty uznaniowe 6 uczniom.

Za zadanie PRZEPUSTOWOŚĆ SIECI (w skrócie PRZEPUSTOWOŚĆ), rozwiązywane w drugim dniu konkursowym 20 kwietnia, przyznawano maksymalnie 100 pkt., w tym nie więcej niż 6 pkt. za rozwiązania uznane za godne szczególnego wyróżnienia. Najwięcej, 94 pkt., otrzymało 6 uczniów, tj. ok. 18%, a 0 pkt. nie otrzymał żaden z uczniów. Jury przyznało punkty uznaniowe 3 uczniom.

Zakończenie I Olimpiady Informatycznej

W dniu 22 kwietnia w Auli Wojewódzkiej Biblioteki Pedagogicznej w Warszawie odbyło się zakończenie I Olimpiady Informatycznej, w trakcie którego ogłoszono wyniki i rozdano nagrody. Nagrody ufundowali: Ministerstwo Edukacji Narodowej oraz firmy IBM Polska, Lexmark, Microsoft i Peryt. Wydawnictwo Naukowe PWN podarowało każdemu uczestnikowi zawodów III stopnia podręcznik w dwóch częściach do nauki elementów informatyki.

Wyniki III etapu I Olimpiady Informatycznej są zawarte w Tabeli 1.

Tabela 1. Wyniki końcowe I Olimpiady Informatycznej.

L.p.	Nazwisko i imię ucznia szkoła	Nagroda i jej fundator
1	2	3
Laureaci I nagrody		
1	Wala Michał I LO im. J. Kasprowicza, Racibórz	komputer 486SX – IBM Polska
2	Stocki Marek XIV LO im. Polonii Belgijskiej, Wrocław	laptop Apple PowerBook – MEN
Laureaci II nagrody		
3	Sobusiak Krzysztof Technikum Elektroniczno-Mechaniczne, Ostrów Wlkp.	drukarka Lexmark 4037 – Lexmark
4	Pawlewicz Jakub III LO, Gdynia	drukarka Lexmark 4212 – Peryt
5	Jarnicki Witold V LO im. A. Witkowskiego, Kraków	Windows 3.1 PL, Word, pakiet CorWin, Sekrety Windows – Peryt
6	Ostrowski Krzysztof IV LO im. T. Kościuszki, Toruń	MS Visual C++ – Microsoft
7	Żarowski Witold XI LO im. M. Reja, Warszawa	Windows 3.1 PL, dBase IV, Sekrety Windows – MEN, Peryt
Laureaci III nagrody		
8	Mucha Marcin II LO im. J. Zamoyskiego, Lublin	dBase IV – MEN
9	Zieliński Piotr VIII LO im. A. Mickiewicza, Poznań	Turbo Pascal 7.0 – MEN
10	Śniady Piotr XIV LO im. Polonii Belgijskiej, Wrocław	Turbo Pascal 7.0 – MEN
11	Błaszczyk Tomasz LO im. S. Żeromskiego, Ozorków	QuattroPro 4.0 – MEN

1	2	3
12	Juraszczyk Szymon I LO im. H. Sienkiewicza, Kędzierzyn-Koźle	QuattroPro 4.0 – MEN
13	Garlewicz Grzegorz II LO im. M. Kopernika, Mielec	QuattroPro 4.0 – MEN
Wyróżnienia		
14	Bogusławski Tomasz XIII LO, Szczecin	WordPerfect, Mouse ECO – MEN
15	Stępiński Juliusz V LO im. Ks. J. Poniatowskiego, Warszawa	WordPerfect, Mouse ECO – MEN
Pozostali uczestnicy zawodów III stopnia		
16	Jakacki Grzegorz XVIII LO im. J. Zamoyskiego, Warszawa	Paradox for DOS – MEN
17	Kaczmarczyk Michał Społeczne LO, Nowy Sącz	Mouse Pad, dyskietki – MEN, Peryt
18	Rzadkowski Paweł II LO im. M. Kopernika, Mielec	Mouse Pad, dyskietki – MEN, Peryt
19	Bobiński Grzegorz IV LO im. T. Kościuszki, Toruń	Mouse Pad, dyskietki – MEN, Peryt
20	Klin Bartosz XXVII LO im. T. Czackiego, Warszawa	Mouse Pad, dyskietki – MEN, Peryt
21	Joniec Radosław II LO im. Jana III Sobieskiego, Kraków	Mouse Pad, dyskietki – MEN, Peryt
22	Jurski Janusz LO im. M. Konopnickiej, Suwałki	Mouse Pad, dyskietki – MEN, Peryt
23	Brzostek Bartosz LO im. Króla Wł. Jagiełły, Dębica	Mouse Pad, dyskietki – MEN, Peryt
24	Klimek Grzegorz VI LO im. J. Kochanowskiego, Radom	Mouse Pad, dyskietki – MEN, Peryt

1	2	3
25	Słocki Jakub II LO im. M. Kopernika, Mielec	Mouse Pad, dyskietki – MEN, Peryt
26	Oksiucik Mirosław III LO, Gdynia	dyskietki – MEN
27	Ambroży Adam Technikum Mechaniczno- -Elektryczne, Chorzów	dyskietki – MEN
28	Olewski Daniel I LO im. E. Dembowskiego, Zielona Góra	dyskietki – MEN
29	Mielańczuk Paweł IX LO im. K. Hoffmanowej, Warszawa	dyskietki – MEN
30	Kopczyński Eryk*) Szkoła Podstawowa nr 143, Warszawa	Sekrety Windows – Peryt
31	Zawirski Michał I LO im. K. Marcinkowskiego, Poznań	dyskietki – MEN
32	Węcel Krzysztof VI LO im. Śniadeckich, Bydgoszcz	dyskietki – MEN
33	Bednorz Adam II LO im. C. K. Norwida, Tychy	dyskietki – MEN

*) Nagroda specjalna dla najmłodszego uczestnika Olimpiady.

Ogłoszono również komunikat o powołaniu reprezentacji Polski na VI Międzynarodową Olimpiadę Informatyczną do Sztokholmu w składzie:

Michał Wala,

Marek Stocki,

Krzysztof Sobusiak,

Jakub Pawlewicz

oraz rezerwowi: Krzysztof Ostrowski i Witold Jarnicki.

Zgodnie z regulaminem Olimpiady, Komitet Główny przyznał nagrody pieniężne następującym nauczycielom za ich wkład w przygotowanie finalistów Olimpiady:

1. mgr inż. Zdzisław Nowakowski ZSE w Mielcu – nauczyciel laureata Grzegorza Garlewicza oraz finalistów Pawła Rzadkowskiego i Jakuba Słockiego.
2. mgr Beata Rudnik III LO w Gdyni – nauczycielka laureata Jakuba Pawlewicza i finalisty Mirosława Oksiucika.
3. mgr inż. Lesław Otręba XIV LO we Wrocławiu – nauczyciel laureata Marka Stockiego.
4. mgr Dzierżysław Malina V LO w Krakowie – nauczyciel laureata Witolda Jarnickiego.
5. mgr Ryszard Kowal II LO w Lublinie – nauczyciel laureata Marcina Muchy.
6. mgr Krzysztof Stefański VIII LO w Poznaniu – nauczyciel laureata Piotra Zielińskiego.
7. mgr Augustyn Kałuża XIV LO we Wrocławiu – nauczyciel laureata Piotra Śniadego.

Komitet Główny Olimpiady zwrócił się do Ministra Edukacji Narodowej z prośbą o przyznanie poza kolejnością po jednym laboratorium komputerowym następującym szkołom, które miały najwięcej uczniów w zawodach III stopnia:

1. II LO im. M. Kopernika, Mielec – 3 uczniów w finale.
2. III LO, Gdynia – 2 uczniów w finale.
3. IV LO im. T. Kościuszki, Toruń – 2 uczniów w finale.
4. XIV LO im. Polonii Belgijskiej, Wrocław – 2 uczniów w finale.

6. ZAWODY MIĘDZYNARODOWE

6.1. Międzynarodowa Olimpiada Informatyczna

W miarę wprowadzania kształcenia informatycznego do szkół, w latach osiemdziesiątych zaczęło wzrastać zainteresowanie konkursami informatycznymi dla młodzieży szkolnej. Niektóre z nich, początkowo lokalne, zaczęły stopniowo mieć zasięg międzynarowodwy, jednak te inicjatywy były dość rozproszone. W tej sytuacji UNESCO, organizacja opiekująca się międzynarodowymi olimpiadami dla młodzieży, podjęła na wniosek Bułgarii decyzję o powołaniu Międzynarodowej Olimpiady Informatycznej (*International Olympiad in Informatics*). Pierwsza olimpiada odbyła się w maju 1989 roku w Prawec w Bułgarii. Od tego czasu zawody odbywają się co roku i cieszą się wielkim zainteresowaniem, a liczba uczestników stale wzrasta.

Celem Międzynarodowej Olimpiady Informatycznej jest zachęcanie młodzieży do:
— głębszego zainteresowania się informatyką i dokładniejszym jej poznaniem,
— sprawdzenia i wykazania swoich umiejętności rozwiązywania zadań za pomocą komputera,
— wymiany wiadomości i doświadczeń z uczniami o zbliżonych zainteresowaniach i umiejętnościach w skali międzynarodowej,
— nawiązywania przyjaznych kontaktów z młodzieżą innych krajów.

Ogólne zasady prowadzenia zawodów, określone w Regulaminie Międzynarodowej Olimpiady Informatycznej, są generalnie zgodne z przyjętymi w ramach UNESCO zasadami prowadzenia międzynarodowych olimpiad z innych przedmiotów.

Kraj organizujący olimpiadę powołuje:
— Przewodniczącego Międzynarodowego Jury,
— Komitet Organizacyjny Olimpiady,
— Komitet Naukowy, odpowiedzialny za przygotowanie i wstępny dobór zadań,
— Komisję Koordynacyjną, odpowiedzialną za ocenę prac zawodników.

Międzynarodowe Jury podejmuje najważniejsze decyzje merytoryczne. W jego skład wchodzą: Przewodniczący, mianowany przez kraj organizatora, oraz

kierownicy wszystkich zespołów narodowych. Do kompetencji Jury należą między innymi:
— ostateczny wybór zadań każdego dnia zawodów ze zbioru propozycji przedstawionych przez Komitet Naukowy,
— podejmowanie decyzji w kwestiach porządkowych i dyscyplinarnych, jeśli takie powstaną w trakcie zawodów,
— rozdział nagród i medali,
— wprowadzanie ewentualnych zmian do regulaminu następnych olimpiad.

Koordynacją przygotowań następnych olimpiad zajmuje się 10-osobowy Komitet Międzynarodowy, wybierany spośród członków Jury.

Każdy kraj zaproszony do udziału w Międzynarodowej Olimpiadzie Informatycznej może przysłać jeden zespół w składzie:
— co nawyżej 4 zawodników, którzy uczęszczali do szkoły w roku poprzedzającym zawody i nie przekroczyli wieku podanego w zaproszeniu,
— kierownik zespołu, który staje się członkiem Międzynarodowego Jury Olimpiady i w czasie zawodów jest odpowiedzialny za tłumaczenie zadań na narodowy język zawodników,
— zastępca kierownika, sprawujący opiekę nad zawodnikami.

W zawodach mogą brać udział tylko zespoły oficjalnie zaproszone – koszty ich zakwaterowania i utrzymania w czasie zawodów pokrywa strona przyjmująca.

Zawodnicy rozwiązują zadania w ciągu dwóch dni, w sesjach trwających po pięć godzin każdego dnia. Zadania są przekazywane w formie pisemnej w narodowym języku zawodnika i w wersji angielskiej, bez żadnych dodatkowych informacji ustnych. W ciągu pierwszej pół godziny od ogłoszenia zadań zawodnicy mogą zadawać pytania na piśmie, na które otrzymują pisemną odpowiedź TAK, NIE albo BEZ ODPOWIEDZI. Każdy zawodnik pracuje samodzielnie, mając do dyspozycji komputer osobisty. Rozwiązywanie zadań nie wymaga stosowania wyspecjalizowanych pakietów oprogramowania ani specjalnych konfiguracji sprzętu. Korzystanie z własnych dyskietek, podręczników i innych pomocy jest zabronione. Zawodnik ma do dyspozycji tylko oprogramowanie zainstalowane w komputerze. Zestaw języków programowania, których można używać, jest ogłaszany rok wcześniej i podawany w zaproszeniu na olimpiadę. Każdy zawodnik może zapoznać się z komputerem i jego oprogramowaniem w dniu poprzedzającym zawody. Ponadto, kilka komputerów ćwiczebnych, które nie są używane w trakcie zawodów, jest dostępnych poza czasem zawodów.

Oceną prac zawodników zajmuje się Komisja Koordynacyjna. Członek Komisji sprawdza po zawodach za pomocą zbioru testów programy będące rozwiązaniami zadań. Sprawdzanie odbywa się w obecności zawodnika i kierownika zespołu. Protokół podpisuje członek Komisji i kierownik zespołu. W przypadku wystąpie-

nia różnicy zdań opisuje się ją w protokole. Decyzję o ostatecznej liczbie punktów przyznanej zawodnikowi za każde zadanie podejmuje Komisja po porównaniu wyników sprawdzania prac wszystkich zawodników. W przypadku pojawienia się poważnych różnic zdań, przekazuje się je do rozpatrzenia przez Jury, które może wypowiedzieć się jedynie w sprawie ogólnych zasad, ale ostateczną propozycję punktacji formułuje zawsze Komisja.

Z przedstawionej przez Komisję Koordynacyjną listy punktów przyznanych zawodnikom (bez nazwisk i krajów) Jury wybiera zawodników, którzy zajęli I, II i III miejsca i otrzymają odpowiednio złote, srebrne i brązowe medale Olimpiady. Od V Olimpiady w Mendozie w 1992 r. najwyższą nagrodą jest puchar ufundowany przez Międzynarodową Federację Przetwarzania Informacji (IFIP – *International Federation of Information Processing*), największą międzynarodową organizację informatyczną, przyznawany temu zawodnikowi lub tym zawodnikom, którzy uzyskają najlepsze wyniki. Puchar IFIP-u przyznaje Jury w osobnym głosowaniu. Jest to nagroda przechodnia – sam puchar przekazuje się po zakończeniu zawodów organizatorom następnej Olimpiady.

Sposób przeprowadzania zawodów olimpiady międzynarodowej zmienianyjest powoli, lecz stale – ostatnio zwiększono liczbę zadań rozwiązywanych każdego dnia zawodów i wprowadzono komputerowo wspomagane sprawdzanie rozwiązań. Można oczekiwać, że zmieniał się będzie również charakter zadań, ale przebiegać to będzie z pewnością wolniej. Organizatorzy Olimpiady Informatycznej w Polsce śledzą te zmiany i tendencje i starają się na bieżąco dostosowywać regulamin i zasady przeprowadzania zawodów krajowych do zmian w przeprowadzaniu olimpiady międzynarodowej. Podobnie postępują organizatorzy regionalnych międzynarodowych konkursów informatycznych, odbywających się przed Olimpiadą Międzynarodową.

Dotychczas odbyły się następujące Międzynarodowe Olimpiady Informatyczne:

Tabela 2. Międzynarodowe Olimpiady Informatyczne.

Rok	Nr	Miasto, państwo	Liczba krajów	Liczba uczestników	Liczba przyznanych medali		
					złote	srebrne	brązowe
1989	I	Pravec, Bułgaria	13	46	6	5	7
1990	II	Mińsk, Białoruś	25	90	8	10	12
1991	III	Ateny, Grecja	23	68	6	14	19
1992	IV	Bonn, RFN	45	166	13	28	41
1993	V	Mendoza, Argentyna	41	155	12	27	42
1994	VI	Sztokholm, Szwecja	49	189	16	36	49

Polscy zawodnicy byli medalistami następujacych olimpiad międzynarodowych:

Mińsk 1990 — Jacek Chrząszcz – srebrny; Rafał Bogacz – brązowy;

Ateny 1991 — Jakub Kruszona – złoty; Jacek Chrząszcz – srebrny; Jacek Pliszka – brązowy;

Bonn 1992 — Tomasz Śmigielski – srebrny; Rafał Bogacz i Jacek Pliszka – brązowe;

Mendoza 1993 — Tomasz Błaszczyk – srebrny; Grzegorz Jakacki – brązowy;

Sztokholm 1994 — Krzysztof Sobusiak – srebrny; Jakub Pawlewicz, Marek Stocki i Michał Wala – brązowe.

6.2. Sprawozdanie z przebiegu Konkursu Informatycznego Krajów Europy Centralnej

W dniach 26–31 maja w Cluj w Rumunii odbył się Konkurs Informatyczny Krajów Europy Centralnej. Konkurs został pomyślany jako zawody regionalne, poprzedzające olimpiadę międzynarodową i stwarzające okazję treningu dla zawodników wytypowanych na olimpiadę. Dodatkowym celem było dokonanie przeglądu koncepcji kształcenia informatycznego w szkołach średnich w Europie Centralnej oraz stworzenie możliwości dalszych dyskusji na temat wybranych zagadnień szczegółowych.

Regulamin konkursu jest zbliżony do regulaminu Międzynarodowej Olimpiady Informatycznej. W konkursie biorą udział zespoły uczniów szkół średnich, po jednym z każdego kraju. W skład każdego zespołu wchodzi co najwyżej 4 uczniów, kierownik zespołu i jego zastępca. Uczniowie muszą uczęszczać do szkoły w roku, w którym odbywa się konkurs. Zawody odbywają się w ciągu dwóch dni, po 3 zadania rozwiązywane indywidualnie każdego dnia. Rozwiązaniem każdego zadania jest program w języku Pascal, C lub QBasic. Zawody jednego dnia trwają 5 godzin. Rozwiązania są sprawdzane na komputerze za pomocą testów. Za rozwiązanie każdego zadania można otrzymać co nawyżej 100 punktów. Ostatecznym wynikiem w konkursie jest suma punktów uzyskanych za wszystkie zadania.

Do udziału w konkursie zostały zaproszone wszystkie kraje Europy Centralnej: Austria, Chorwacja, Czechy, Polska, Republika Federalna Niemiec, Słowacja, Słowenia i Węgry. Z zaproszenia skorzystały zespoły z Chorwacji, Czech, Polski,

Rumunii i Węgier. Ponadto, gospodarze konkursu mieli prawo zaprosić inne kraje – w tym charakterze wystąpiły zespoły z Jugosławii, Mołdawii i Turcji.

Sklasyfikowano 31 zawodników. Trzech pierwszych (po jednym zawodniku z Rumunii, Czech i Węgier) otrzymało złote medale, siedmiu (w tym trzech Polaków – Krzysztof Sobusiak, Marek Stocki i Michał Wala) – srebrne, ośmiu (w tym jeden Polak – Jakub Pawlewicz) – brązowe. Nie prowadzono klasyfikacji zespołowej.

6.3. Sprawozdanie z przebiegu VI Międzynarodowej Olimpiady Informatycznej

VI Międzynarodowa Olimpiada Informatyczna odbyła się w dniach 3–10 lipca w Hanninge – gminie Sztokholmu, leżącej na południe od centrum miasta. Polskę reprezentowali laureaci krajowej I Olimpiady Informatycznej: Michał Wala, Marek Stocki, Krzysztof Sobusiak i Jakub Pawlewicz. Przewodniczącym delegacji był prof. dr hab. inż. Stanisław Waligórski, a jego zastępcą – dr Andrzej Walat.

Zawody odbywały się w nowoczesnym budynku KTH – Królewskiej Szkoły Technicznej – jednej z najlepszych uczelni szwedzkich. Polegały one na samodzielnym rozwiązywaniu zadań za pomocą komputera. W ciągu dwóch pięciogodzinnych sesji we wtorek 5 lipca i w czwartek 7 lipca należało rozwiazać łącznie sześć zadań.

W zawodach uczestniczyło 189 zawodników z 49 krajów. Nie prowadzono klasyfikacji zespołowej. Złote medale otrzymali zawodnicy: z Rosji i Chin – po trzech, z Niemiec i Węgier – po dwóch oraz po jednym z Białorusi, Czech, Korei Płd., Słowacji i USA. Przechodni Puchar IFIP-u otrzymał Viktor Bargatc z Rosji, który zdobył 195 punktów na 200 możliwych i wyprzedził następnego zawodnika aż o 25 punktów. Polski zespół wypadł lepiej niż w dwóch ostatnich olimpiadach w Niemczech i Argentynie. Wszyscy polscy zawodnicy zdobyli medale: Krzysztof Sobusiak – srebrny, a Jakub Pawlewicz, Marek Stocki i Michał Wala – brązowe.

W czasie olimpiady zorganizowano kilka atrakcyjnych wycieczek: do firmy Ericsson – potentata w dziedzinie elektroniki i telekomunikacji, tramwajem wodnym po Sztokholmie, do muzeum statku Waza i statkiem na wyspę Uto. Odbyły się również interesujące seminaria, na których przedstawiono nowe narzędzia informatyczne np. programy do projektowania i animacji Alias i Softimage oraz Turbo Pascal 8 Borlanda.

VII Międzynarodowa Olimpiada Informatyczna odbędzie się na przełomie czerwca i lipca 1995 roku w Eindhoven w Holandii.

7. TEKSTY I ROZWIĄZANIA ZADAŃ I OLIMPIADY INFORMATYCZNEJ

W tym rozdziale zamieszczamy teksty zadań I 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 KOMENTARZA. W jednym przypadku (zob. Omówienie rozwiązań podanych przez uczniów w p. 7.2.2) pytania uczniów spowodowały uściślenie treści zadania przez Komitet i ogłoszenie uzupełnienia wszystkim uczniom biorącym udział w zawodach II 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. Główną częścią rozwiązania są fragmenty programów realizujących przedstawione metody. Najczęściej są to procedury odpowiadające poszczególnym krokom algorytmu rozwiązującego zadanie. Te fragmenty programów, ze względu na ograniczoność miejsca, na ogół nie zawierają szczegółowej obsługi błędów wejścia/wyjścia. Z jednym wyjątkiem programy są napisane w języku Turbo Pascal – w rozwiązaniu zadania ANAGRAMY autor

umieścił program napisany w języku C. Uzasadnieniem tego wyboru może być fakt, że około 10% uczniów napisało program rozwiązujący to zadanie właśnie w języku C.

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.

7.1. Zawody I stopnia

7.1.1. Zadanie TRÓJKĄTY (Autor: Piotr Chrząstowski-Wachtel)

Treść zadania TRÓJKĄTY

Dany jest skończony, co najmniej trzyelementowy zbiór A odcinków o długościach wymiernych. Chcemy zbadać, czy z każdych trzech odcinków z tego zbioru można zbudować trójkąt.

Zestaw danych wejściowych jest co najmniej trzyelementowym ciągiem długości wszystkich odcinków ze zbioru A ułożonych w jakiejś kolejności. Każda długość odcinka (liczba wymierna) jest zapisana w postaci *licznik*/*mianownik*, gdzie *licznik* oraz *mianownik* są dodatnimi liczbami całkowitymi, nie większymi niż 9999.

Kolejne długości w tym ciągu są oddzielone odstępem lub pojedynczym znakiem końca wiersza.

Należy otrzymać odpowiedź:
TAK, jeśli z każdej trójki odcinków można zbudować trójkąt.
NIE, jeśli nie z każdej trójki odcinków można zbudować trójkąt.
NONSENS, jeśli zestaw danych jest niepoprawny, to znaczy nie spełnia podanych wyżej warunków.

Przykłady

Dla zestawu danych wejściowych:

13/10 1/2 6/5 11/6 9/7 3/5 9/7 13/10 9/5 8/5

odpowiedź brzmi NIE, bo na przykład nie da się zbudowować trójkąta z odcinków o długościach: 6/5 3/5 9/5.

Dla zestawu danych wejściowych:

1/2 3/5 2/3 4/7 1/1 4/6

odpowiedź brzmi TAK.

Dla zestawu danych wejściowych:

1/2 3/5 2/3 4/7 1 4/6

odpowiedź brzmi NONSENS, bo 1 nie jest parą liczb przedzielonych znakiem /.

Zadanie

Ułóż program, który kolejno dla każdego zestawu danych z pliku TKT.IN generuje właściwą odpowiedź: TAK, NIE albo NONSENS i zapisuje ją w pliku TKT.OUT.

Tekst programu w postaci źródłowej powinien być zapisany w pliku o nazwie TKT.???, gdzie zamiast ??? wstawia się ciąg liter właściwy dla użytego języka programowania.

Program w postaci wykonywalnej powinien mieć nazwę TKT.EXE.

Rozwiązanie zadania TRÓJKĄTY

Już pobieżna analiza problemu pozwala zauważyć, że nie trzeba sprawdzać wszystkich trójek liczb, a jedynie dwie najmniejsze i największą. Oznaczmy przez $Min1$ i $Min2$ długości dwóch najkrótszych odcinków, a przez Max – długość najdłuższego odcinka w zbiorze A (z założenia zbiór A zawiera co najmniej trzy elementy). Prawdziwy jest następujący fakt:

Twierdzenie. *Z dowolnie wybranej trójki odcinków ze zbioru A można zbudować trójkąt wtedy i tylko wtedy, gdy spełniona jest następująca nierówność:*

$$Min1 + Min2 > Max,$$

zwana **nierównością trójkąta** *dla liczb $Min1$, $Min2$ i Max.*

Dowód. Jeżeli $Min1 + Min2 \le Max$, to z odcinków o tych długościach nie da się zbudować trójkąta. W przeciwnym natomiast przypadku liczby $Min1$, $Min2$, Max spełniają warunek trójkąta:

$$Min1 + Min2 > Max$$

i wtedy dla dowolnej innej trójki odcinków o długościach odpowiednio a, b i c takich, że $a \le b \le c$ mamy

$$a + b \ge Min1 + Min2 > Max \ge c,$$

gdzie pierwsza i ostatnia nierówność wynika z definicji *Min*1, *Min*2 i *Max*. Zatem $a + b > c$, czyli z odcinków o długościach a, b, c można również zbudować trójkąt.

Aby znaleźć dwie najmniejsze i największą liczbę w ciągu można, wczytawszy kolejny ułamek z danych wejściowych, wykonać tylko dwa porównania. Załóżmy, że wczytujemy ciąg liczb wymiernych $a_1, \ldots, a_n$, w którym liczba elementów, chociaż skończona, nie jest ograniczona. Niech *Min*1 będzie najmniejszą, *Min*2 – następną po niej co do wielkości, zaś *Max* – największą liczbą z wczytanego dotąd ciągu $a_1, \ldots, a_k$ $(2 < k < n)$. Biorąc element a_{k+1} wystarczy porównać go z *Min*2 i w przypadku gdy $a_{k+1} < Min2$ – porównać go jedynie z *Min*1, a gdy $a_{k+1} \geq Min2$ – porównać go z *Max*, dokonując w miarę potrzeby odpowiednich modyfikacji wartości tych zmiennych. Pierwsze trzy liczby ciągu wczytujemy oczywiście poza pętlą i przypisujemy odpowiednim zmiennym.

Warunek trójkąta można by oczywiście sprawdzać na bieżąco i przerwać wykonywanie algorytmu po napotkaniu pierwszej trójki, która warunku nie spełnia. Takie sprawdzanie warto wykonać jedynie wtedy, gdy któraś ze zmiennych *Min*1, *Min*2 i *Max* zmieniła swoją wartość. To dodatkowe sprawdzanie warunku trójkąta zwiększa jednak koszt (złożoność) algorytmu, gdy dane wejściowe dają odpowiedź TAK.

W przedstawionym poniżej fragmencie programu w języku Pascal pomijamy wczytywanie danych i zakładamy, że opisana została procedura `KolejnaDana(x)`, która zmiennej `x` nadaje wartość wczytanej danej (czyli ułamka reprezentującego liczbę wymierną) i ustala jednocześnie wartość zmiennej logicznej `SaJeszczeDane` na `True` wtedy i tylko wtedy, gdy nie była to jeszcze ostatnia dana w ciągu. Zakładamy również, że dysponujemy funkcją logiczną `Mniejszy(a,b)`, która stwierdza, czy ułamek `a` jest mniejszy od ułamka `b`. Ponadto zakładamy, że nastąpiła odpowiednia inicjalizacja zmiennych `Min1`, `Min2` i `Max` oraz zmiennej `SaJeszczeDane` po wczytaniu trzech pierwszych danych: `Min1 ≤ Min2 ≤ Max`.

```
while SaJeszczeDane do begin
 KolejnaDana(x);
 if Mniejszy(x,Min2) then
  if Mniejszy(x,Min1) then begin
   Min2:=Min1;  Min1:=x
  end
  else Min2:=x
   {*}
 else {x≥Min2}
```



```
  if Mniejszy(Max,x) then
   Max:=x
  {*}
end;
```

Zauważmy, że nie musimy zmieniać wartości zmiennych w przypadku, gdy `Min2 ≤ x ≤ Max`.

W miejscach zaznaczonych gwiazdką można ewentualnie sprawdzać na bieżąco spełnienie warunku trójkąta dla nowych wartości `Min1`, `Min2` lub `Max` i w razie niespełnienia – przerwać dalsze przetwarzanie danych, choćby przez ustawienie wartości zmiennej `SaJeszczeDane` na `False`.

Funkcja `Mniejszy(a,b)` jest funkcją o wartościach logicznych, stwierdzającą czy ułamek `a` jest mniejszy od ułamka `b`. Zauważmy, że zastąpienie tej funkcji przez proste porównanie zmiennych, (np. `x < Min1`) jest możliwe tylko przy założeniu, że reprezentujemy liczby wymierne przez któryś ze standardowych typów z określoną relacją „<". Użycie takiej reprezentacji (np. standardowych typów rzeczywistych) do reprezentowania liczb wymiernych nieuchronnie prowadzi jednak do powstawania błędów zaokrągleń, które mogą wypaczać wyniki – było to przyczyną błędów w większości rozwiązań podanych przez uczniów.

Jak zatem wybrać odpowiednią reprezentację? Wydaje się, że jedynym sensownym rozwiązaniem jest pamiętanie ułamków w postaci par liczb naturalnych. Należy wtedy samemu opisać sposób porównywania i dodawania ułamków. Dla przykładu, jeżeli mamy dwa ułamki: $u_1 = (l_1, m_1)$ i $u_2 = (l_2, m_2)$, to sprawdzenie, czy $u_1 < u_2$ sprowadza się do rozstrzygnięcia, czy $l_1 m_2 < l_2 m_1$. Obliczanie takich iloczynów może jednak łatwo wyprowadzić np. poza prosty typ `Integer` w Turbo Pascalu, gdyż $9999 * 9999 > 2^{16}-1$. Użycie innego, niestandardowego typu całkowitego (np. `LongInt`) również nie jest w pełni zadawalającym rozwiązaniem, gdyż w typowych realizacjach języków programowania brak jest typu całkowitego zdolnego reprezentować liczby 14-cyfrowe, a do takich za chwilę dojdziemy. Aby sprawdzić bowiem, czy suma dwóch ułamków (l_1, m_1) oraz (l_2, m_2) jest większa niż (l_3, m_3) należy stwierdzić, czy $(l_1 m_2 + l_2 m_1)\, m_3 > l_3 m_1 m_2$. Iloczyn z prawej strony może się jednak okazać liczbą 14-cyfrową (po lewej stronie możemy nawet otrzymać liczbę 15-cyfrową, ale dla sprawdzenia warunku trójkąta wystarczy, jeśli od $l_3 m_1 m_2$ odejmiemy $l_1 m_2 m_3$ i sprawdzimy, czy wynik jest mniejszy od $l_2 m_1 m_3$, a wszystkie te liczby są co najwyżej 14-cyfrowe).

Widzimy więc, że liczby, z którymi możemy mieć do czynienia w tym zadaniu są na tyle duże, że mogą przekroczyć zakresy standardowych typów danych.

Wyjścia z tej sytuacji są dwa. Należy albo znaleźć odpowiednio bogaty typ danych, albo samemu zadbać o realizację odpowiedniej arytmetyki. W Turbo Pascalu 6.0 istnieje np. typ Comp, który co prawda jest typem rzeczywistym, ale reprezentuje dokładnie wszystkie wartości całkowite aż do liczb osiemnastocyfrowych. Jest on więc wystarczający dla naszych celów. Można także użyć w tym celu typu Extended, który ma podobne możliwości.

Zadbanie samemu o arytmetykę oznacza reprezentowanie liczb w postaci tablic cyfr i napisanie własnych funkcji porównujących i dodających tak reprezentowane liczby. Rozwiązanie to jest na tyle uniwersalne, że nawet gdyby w zadaniu była mowa o znacznie większym zakresie liczb niż [1, 9999], to zmiana w programie polegałaby jedynie na zmianie odpowiedniej stałej oznaczającej maksymalną liczbę cyfr potrzebnych do reprezentowania liczników i mianowników. Rzecz jasna taka reprezentacja liczb zajmuje większą pamięć, ale wymagania naszego programu na pamięć są niewielkie i sprowadzają się do pamiętania jednocześnie co najwyżej 4 liczb. Niewątpliwie możemy sobie pozwolić na taką rozrzutność, jak własna arytmetyka.

Dla przykładu, typowa reprezentacja liczb wymiernych na potrzeby naszego zadania mogłaby wyglądać następująco:

```
const
 Baza=10; {podstawa ukladu pozycyjnego}
 n   =4;  {maksymalna liczba cyfr występujących
           w liczniku i mianowniku}
 nn  =8;  {nn=2*n}
type
 Nat   =array[1..n] of Integer; {tablica cyfr liczby}
 Ulamek=record  l,m:Nat  end;

function Mmniejszy(x,y:Ulamek):Boolean;
 type NatNat:array[1..nn] of Integer;
 var i        :Integer;
     xlym,xmyl:NatNat;
 procedure Iloczyn(x,y:Nat;var z:NatNat);
  {Procedura Iloczyn oblicza metoda slupkowa iloczyn dwóch
   liczb naturalnych x i y zapisanych w postaci tablic cyfr
   i wynik umieszcza w tablicy z. Dzialania sa wykonywane
   w ukladzie pozycyjnym o podstawie Baza.}
  var i,k:Integer;
```



```
 begin
  for i:=1 to nn do z[i]:=0;
  for i:=1 to n do
   for k:=1 to n do begin
    z[i+k-1]:=z[i+k-1]+(x[i]*y[k] mod Baza);
    z[i+k]:=z[i+k]+z[i+k-1] div Baza+x[i]*y[k] div Baza
   end
 end; {Iloczyn}
begin
 Iloczyn(x.l,y.m,xlym);
 Iloczyn(x.m,y.l,xmyl);
  {Sprawdzamy, która z tablic xlym i xmyl reprezentuje
   mniejsza liczbe.}
 i:=nn; {najbardziej znaczaca cyfra ma indeks 2n}
 while (i>=1) and (xmyl[i]=xlym[i]) do i:=i-1;
  {Zakladamy, ze wartosc warunku logicznego jest obliczana
   „leniwie", tzn. jesli i<1, to nie jest obliczana druga
   czesc koniunkcji.}
 Mniejszy:=(i>=1) and (xmyl[i]<xlym[i])
end; {Mniejszy}
```

Funkcje: dodającą dwie liczby i porównującą wynik z trzecią konstruujemy podobnie.

Omówienie rozwiązań podanych przez uczniów

Zadanie o trójkątach sprawiło uczniom najwięcej kłopotu. Ani jedno rozwiązanie nie było całkowicie poprawne. Najwyżej oceniony program uzyskał 87 punktów na 100 możliwych. Zrozumienie treści zadania nie sprawiło kłopotów interpretacyjnych, ale prawie wszystkim uczniom umknęła uwadze trudność związana z niedokładnością reprezentacji liczb wymiernych w komputerze. W większości rozwiązań wartości ułamków były przeliczane na któryś z typów rzeczywistych (Real, Float, ...) i działania wykonywano na liczbach tego typu. Prowadziło to nieuchronnie do powstawania zaokrągleń, które dawały z reguły błędne wyniki. Kto nie wierzy, niech sprawdzi na przykład, że 1/3 + 2/3 > 1 w większości języków programowania.

W momencie odkrycia, że należy znaleźć tylko dwie najmniejsze liczby i największą, większość uczniów poprzestała na tym i nie próbowała już dalej optymalizować działania algorytmu. Każdy wczytany element porównywano ze wszystkimi trzema pamiętanymi wartościami, nie zauważając możliwości

uniknięcia jednego porównania, jeżeli zaczniemy od środkowego elementu. Ten prosty zabieg w przypadku ciągu losowego przyspiesza działanie algorytmu o ok. 30% w porównaniu z algorytmem, który zaczyna porównania od najmniejszej, poprzez średnią (te dwa porównania w większości przypadków dają wynik korzystny dla elementu), kończąc na największej z liczb.

Poza nielicznymi rozwiązaniami większość uczniów przeszła do porządku dziennego nad problemem niedokładnej reprezentacji liczb wymiernych w komputerze. Używając któregoś z dostępnych typów rzeczywistych przybliżano zadane wartości wymierne najbliższymi liczbami rzeczywistymi reprezentowalnymi w komputerze, też siłą rzeczy liczbami wymiernymi, na ogół różnymi od nich. Jedynie bowiem te liczby wymierne, których mianowniki są potęgami dwójki mają szansę być dokładnie reprezentowane.

Jeden z uczniów wpadł na pomysł rozbicia liczby na dwie połowy po 9 cyfr, każda z nich reprezentowana w typie `LongInt` – w rzeczywistości wykonywał on odpowiednie działania w systemie o podstawie 2^9.

Niektóre rozwiązania uczniów działały zgodnie ze schematem:

— wczytaj dane,

— przetwórz dane,

— wypisz wynik.

Jak pokazalismy w naszym rozwiązaniu, możliwe jest przemieszanie fazy 1 z fazą 2 – wtedy nie trzeba przechowywać w pamięci całego ciągu. Użycie powyższego schematu powodowało przepełnienie pamięci w przypadku długiego pliku z danymi wejściowymi (np. takiego, jak w testach nr 8 i 9).

Testy

Testy, na których sprawdzano rozwiązania uczniów były czterech rodzajów.

1. Testy podstawowe, sprawdzające poprawność algorytmu dla prostych danych. Chodziło tu przede wszystkim o zbadanie, czy program radzi sobie w najprostszych sytuacjach.

Testy 1 i 2: kilkuelementowe ciągi z odpowiedzią TAK i NIE.

Testy 3 i 4: ciągi posortowane rosnąco – z odpowiedzią TAK i malejąco – z odpowiedzią NIE.

Test 5 – ciąg, którego wszystkie elementy są równe, z odpowiedzią TAK:
1/3 2/6 3/9 4/12 5/15 6/18 7/21 8/24 9/27 10/30 11/33 12/36 13/39 14/42 15/45

Testy 6 i 7: ciągi trzyelementowe z odpowiedzią TAK i NIE.

2. Testy złożoności, wychwytujące algorytmy o dużej złożoności, zarówno czasowej, jak i pamięciowej.

Test 8 – badanie złożoności czasowej z odpowiedzią TAK:
3/2 4/3 5/4 ... 1001/1000

Test 9 – badanie złożoności pamięciowej z odpowiedzią TAK:
1/3 2/6 ... 3333/9999
1/4 2/8 ... 2499/9996
1/5 2/10 ... 1999/9995
1/3 2/6 ... 3333/9999 i tak dalej – plik zajmował 1.5 MB pamięci.

3. Testy implementacji, sprawdzające czy użyte typy danych były odpowiednie – były to testy najtrudniejsze dla większości programów.

Test 10 sprawdzał poprawność implementacji liczb wymiernych z odpowiedzią NIE: 1/3 2/3 1/1

Test 11 sprawdzał przekroczenie zakresu typu `LongInt` z odpowiedzią TAK:
1/9999 2/9998 1/9997

4. Testy wejścia/wyjścia, badające zdolność programu do odpowiedniego reagowania w sytuacjach błędnych danych wejściowych.

Test 12 badał reakcję na niekompletny element danych (NONSENS): 1/2 4/2 1/ 3/2.

Test 13 badał przekroczenie zakresu danych (NONSENS): 14/4 67/2 3/0 8/3.

Test 14 badał reakcję na zbyt małą liczbę elementów: 1/2 2/3.

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

Treść zadania SPONSOR

Pływak Daniel Waterproof jest od dzieciństwa sponsorowany przez swojego wuja, bogatego biznesmena. Na zakończenie kariery pływackiej wuj postanowił ufundować mu specjalną nagrodę. Daniel przechowywał w komputerze wszystkie kolejne czasy uzyskane zarówno na treningach, jak i na zawodach w swojej koronnej konkurencji – 50 m stylem dowolnym, więc wuj wymyślił, że wysokość nagrody będzie uzależniona od długości najdłuższego malejącego ciągu wyników, jaki uda się wyszukać w kolejnych danych. Wysokość nagrody w dolarach będzie stokrotnie większa od długości tego ciągu.

Zestaw danych wejściowych jest niepustym ciągiem kolejno uzyskanych wyników w ciągu całej kariery Daniela. Wyniki mierzy się w sekundach z dokładnością do tysięcznych części sekundy. Każdy wynik jest liczbą z domkniętego przedziału [20,30], mającą dokładnie trzy cyfry po przecinku (kropce dziesiętnej). Kolejne wyniki oddziela się odstępem lub pojedynczym znakiem końca wiersza.

Odpowiedzią jest liczba całkowita równa wysokości nagrody w dolarach lub słowo NONSENS, jeśli zestaw danych jest niepoprawny, to znaczy nie spełnia wyżej podanych warunków.

Przykład

Dla zestawu danych wejściowych:

22.155 23.252 22.586 22.450 23.372 22.228 22.087 23.091 22.190
22.140 22.057 22.123 22.359 22.190 22.140 22.523 22.384 22.488
23.201 23.050

zawierającego dwadzieścia kolejnych wyników odpowiedzią jest liczba 700 ze względu na to, że maksymalna długość malejącego ciągu wyników wynosi 7. Można wskazać nawet dwa takie malejące ciągi o długości 7. Tworzą je wyniki:

2-gi, 3-ci, 4-ty, 6-ty, 9-ty, 10-ty i 11-ty:
23.252 22.586 22.450 22.228 22.190 22.140 22.057,
oraz
2-gi, 3-ci, 4-ty, 6-ty, 9-ty, 10-ty i 12-ty:
23.252 22.586 22.450 22.228 22.190 22.140 22.123.

Dla zestawu danych wejściowych:

22 22.4357 22.5

odpowiedzią jest słowo NONSENS, bo każdy z trzech wyników został zapisany niezgodnie z podanymi zasadami.

Zadanie

Ułóż program, który kolejno dla każdego zestawu danych z pliku SPO.IN generuje właściwą odpowiedź: liczbę całkowitą lub słowo NONSENS i zapisuje ją w pliku SPO.OUT.

Tekst programu w postaci źródłowej powinien być zapisany w pliku o nazwie SPO.??? gdzie zamiast ??? wstawia się ciąg liter właściwy dla użytego języka programowania.

Program w postaci wykonywalnej powinien mieć nazwę SPO.EXE.

Rozwiązanie zadania SPONSOR

Zadanie to, w odróżnieniu od zadania o trójkątach, umożliwia pisanie algorytmów bardzo kosztownych. Sprowadza się bowiem do problemu znalezienia w ciągu malejącego podciągu o maksymalnej długości, a liczba podciągów danego ciągu rośnie wykładniczo ze wzrostem jego długości (ich liczba jest równa 2^n dla ciągu długości n).

Na początku warto więc zastanowić się nad możliwą długością ciągu wejściowego i przynajmniej oszacować rząd tej wielkości. Kariera pływaka trwa zazwyczaj kilkanaście lat, zapewne nie dłużej niż 20, ale i nie krócej niż, powiedzmy, rok. W ciągu roku zawodnik odbywa zwykle około 100–300 treningów i na jednym treningu parokrotnie przepływa dystans 50 m z pomiarem czasu. Należało się więc spodziewać, że plik wejściowy może zawierać co najmniej kilkaset, a zapewne kilka tysięcy wyników. Można również przyjąć, że rozsądnym ograniczeniem górnym długości ciągu wejściowego w tym zadaniu jest 100000. Trudno sobie wyobrazić bowiem, aby Daniel codziennie zapisywał swoje czasy po więcej niż 10 na każdym treningu, przez ponad 20 lat.

Widać więc, że przejrzenie wszystkich podciągów, jako metoda rozwiązywania, nie wchodzi tutaj w rachubę, ze względu na jej złożoność. W zasadzie należy szukać algorytmu o złożoności rzędu co najwyżej n^2, gdzie n jest długością ciągu danych – dla złożoności wyższego rzędu algorytm działałby nieznośnie długo (organizatorzy olimpiady tak zwykle formułują zadania, aby ich rozwiązanie dało się sprawdzić w rozsądnym czasie).

Załóżmy, że dane w pliku SPO.IN reprezentują ciąg $a_1, a_2, \ldots, a_n$. Rozwiązanie zadania sprowadza się do wyszukania najdłuższego (niekoniecznie spójnego) podciągu malejącego w tym ciągu. Ze względu na wykładniczą liczbę podciągów danego ciągu, wykluczamy rozwiązania w rodzaju „zbadaj wszystkie podciągi i wybierz najdłuższy".

Kluczem do znalezienia efektywnego algorytmu jest spostrzeżenie, że w trakcie przetwarzania kolejnych wyrazów ciągu nie musimy pamiętać liczb, które mają małe szanse znaleźć się w najdłuższym podciągu malejącym. Które liczby mają tę własność?

Żeby odpowiedzieć na to pytanie zastanówmy się, jaki najdłuższy podciąg $b_1, b_2, \ldots, b_j$ można przedłużyć malejąco za pomocą liczby a_i. Widać, że krótszych niż najdłuższy z podciągów nie ma co rozważać: spośród dwóch ciągów zakończonych przez a_i interesuje nas tylko ten dłuższy.

Może zajść zatem jeden z trzech przypadków.

1. Jeśli liczba a_i jest największą spośród wczytanych do tej pory liczb $a_1, \ldots, a_i$, to z racji swojej wielkości może zacząć najdłuższy malejący ciąg (czyli przedłużyć podciąg pusty).
2. Jeżeli a_i jest najmniejszą liczbą wśród liczb $a_1, \ldots, a_i$, to wydłużamy najdłuższy podciąg malejący $b_1, \ldots, b_j$, dopisując a_i na jego końcu.
3. Jeśli natomiast istnieje taka liczba a_k, że $a_k < a_i$ oraz a_k kończy najdłuższy podciąg malejący $b_1, \ldots, b_{j-1}, a_k$ długości j i $b_{j-1} > a_i$, to liczba a_k przestaje być dla nas interesująca, jako zakończenie takiego podciągu. Bowiem zamia-

na a_k na a_i w tym podciągu jest o tyle korzystna, że łatwiej będzie wydłużyć podciąg $b_1, \ldots, b_{j-1}, a_i$ niż $b_1, \ldots, b_{j-1}, a_k$.

Bezpośrednio tuż po uwzględnieniu liczby a_i jest ona zawsze istotna: albo wydłuża najdłuższy, znaleziony do tej pory podciąg malejący, albo powoduje usunięcie któregoś z mniejszych (lub równych – to dla ujednolicenia algorytmu) elementów z końca ciągu malejącego.

Utworzymy w naszym programie tablicę `t[1..10001]` o wartościach typu rzeczywistego, dla której po wczytaniu liczby a_i jest zachowany niezmienniczo następujący warunek: jeżeli w ciągu $a_1, \ldots, a_i$ istnieje podciąg malejący długości `k`, to element `t[k]` jest największą liczbą spośród tych, które kończą podciągi długości `k`. Jeżeli podciąg malejący długości `k` nie istnieje, to element `t[k]` ma wartość nieokreśloną. Zakres tablicy `t` wynosi `10001` i wynika z długości najdłuższego podciągu malejącego, jaki można utworzyć z różnych wyników uzyskanych przez Daniela (przypomnijmy, wyniki Daniela są z przedziału domkniętego [20,30] i ich wartości mają trzy cyfry po kropce).

Przyjmijmy, że wartością zmiennej `Najdl` typu całkowitego jest długość najdłuższego podciągu malejącego, utworzonego spośród dotychczas wczytanych elementów. Jest to jednocześnie największy indeks elementu w tablicy `t`, zawierającego określoną wartość. Zapiszemy teraz algorytm przetworzenia kolejnej liczby `a` z danych wejściowych. Znaczenie zmiennej `SaJeszczeDane` i działanie procedury `KolejnaDana` są identyczne jak w rozwiązaniu zadania TRÓJKĄTY (zob. p. 7.1.1).

```
KolejnaDana(a);
t[1]:=a;  Najdl:=1;
while SaJeszczeDane do begin
 KolejnaDana(a);
 if a<t[Najdl] then begin
  Najdl:=Najdl+1;
  T[Najdl]:=a
 end
 else begin
  znajdz najmniejsze j takie, ze t[j]≤a;
  t[j]:=a
 end
end
```

Zastanówmy się teraz nad realizacją polecenia:

```
znajdz najmniejsze j takie, ze t[j]≤a;
```


Okazuje się, że działanie algorytmu można przyspieszyć – wystarczy zauważyć, że elementy w tablicy `t` występują w porządku malejącym. Aby to uzasadnić, przypomnijmy sobie znaczenie tablicy `t` i rozważmy, co by oznaczało, gdyby po rozpatrzeniu elementów $a_1, a_2, \ldots, a_i$, dla pewnej pary indeksów `k<l` było `t[k]≤t[l]`? Znaczyłoby to, że w ciągu $a_1, \ldots, a_i$ najdłuższy pociąg długości `k` kończy się liczbą nie większą, niż dłuższy od niego malejący podciąg o długości `l`. Ale wtedy pierwszych `k` wyrazów tego drugiego podciągu malejącego kończyłoby się liczbą większą niż `t[l]`, a zatem również większą od `t[k]`. Otrzymaliśmy więc sprzeczność z założeniem, że `t[k]` jest największym elementem kończącym podciąg malejący długości `k`. Elementy tablicy `t` są zatem w porządku malejącym.

Dzięki tej własności, do lokalizacji indeksu `j` w tablicy `t` możemy zastosować algorytm wyszukiwania binarnego, którego złożoność jest rzędu $\log_2 m$, jeśli przeszukujemy odcinek tablicy `t` między `1` i m.

Następująca funkcja jest zatem realizacją powyższego polecenia:

```
function Znajdz(a:Real):Integer;
 {Funkcja znajduje najmniejszy indeks j w tablicy t taki,
  ze t[j]≤a. Zakladamy, ze elementy w tablicy t sa
  uporzadkowane malejaco oraz ze t[Najdl]≤a. Stosujemy
  algorytm wyszukiwania binarnego.}
 var l,p,s:Integer;
begin
 l:=0; p:=Najdl+1;
 repeat
  s:=(l+p) div 2;
  if a<t[s] then l:=s
  else p:=s
 until l=p-1;
 Znajdz:=p
end; {Znajdz}
```

Aby udowodnić poprawność algorytmu tworzenia tablicy `t` pokażemy, że tablica ta zawiera właściwe wartości po każdym wykonaniu pętli `while`. Zastosujemy indukcję względem wskaźnika tablicy `t`. Oznaczmy przez a_i daną wczytaną przez procedurę `KolejnaDana` w jej i-tym wywołaniu.

Zauważmy najpierw (podstawa indukcji), że po inicjalizacji tablica `t` zawiera istotnie jedyną właściwą wartość – wczytaliśmy jeden element a_1 i jedyny malejący podciąg ma długość `1` i kończy się elementem a_1.

Załóżmy indukcyjnie, że po wczytaniu $i-1$ kolejnych elementów ciągu wejściowego, element tablicy `t[k]` zawiera największy element kończący malejący podciąg ciągu $a_1, a_2, \ldots, a_{i-1}$ długości `k` dla `k=1,2,...Najdl`. Wykażemy, że po rozpatrzeniu elementu a_i warunek ten jest nadal spełniony dla ciągu $a_1, \ldots, a_{i-1}, a_i$.

Jeżeli element a_i jest najmniejszym z dotychczas wczytanych, to nie może on zmienić żadnej z istniejących wartości `t[1],...,t[Najdl]`, gdyż wszystkie są od niego większe, natomiast dopisując go na końcu malejącego podciągu długości `Najdl`, zakończonego większym od niego elementem `t[Najdl]`, otrzymujemy podciąg malejący długości `Najdl+1`, zakończony przez a_i. Odpowiadają temu przypadkowi przypisania w instrukcji `if` po `then`.

W przypadku, gdy a_i nie jest najmniejszym elementem wśród dotychczas wczytanych, jeżeli `j` jest najmniejszym indeksem spełniającym `t[j]`$\le a_i$, to możliwe są dwie sytuacje. Jeżeli `j=1`, to znaczy, że została wczytana największa z dotychczas wprowadzonych liczb i przypisanie `t[1]:=`a_1 jest jak najbardziej uzasadnione. Istotnie, największym elementem kończącym jednoelementowy podciąg malejący jest właśnie a_i i jednocześnie a_i nie może kończyć żadnego innego ciągu malejącego.

Jeżeli natomiast `j>1`, to `t[j-1]`$> a_i \ge$`t[j]`. W tym przypadku najdłuższy podciąg malejący długości `j-1` jest zakończony elementem większym niż a_i, więc po dopisaniu na końcu tego podciągu elementu a_i, otrzymujemy pociąg malejący długości `j`, niezależnie od tego, czy a_i jest równe `t[j]`, czy jest od niego ostro większe (w tym drugim przypadku istotnie poprawiamy „jakość" tego podciągu). Oczywiście moglibyśmy sprawdzać przed przypisaniem czy `t[j]`$<> a_i$, ale przypisania są mało kosztowne i nie opłaca się tego robić.

Metoda dowodzenia zastosowana w tym zadaniu nazywa się **metodą niezmienników Hoare'a** (czytaj Hora). Jest ona na tyle ważna, że warto podać ją tu w pełnej postaci, jako ogólną metodę dowodzenia **częściowej poprawności pętli** (częściowej, czyli takiej, że interesujący nas warunek końcowy zostanie spełniony o ile pętla zakończy swoje działanie).

Jeżeli mamy pętlę w postaci

`while` B `do` I;

gdzie B jest warunkiem logicznym, zaś I – instrukcją (złożoną), to pokazanie, że pętla wykonuje to o co nam chodzi rozpoczynamy od ustalenia dwóch warunków P i K opisujących opowiednio sytuację początkową (tuż przed wejściem do pętli) i końcową (czyli żądaną przez nas własność danych po wykonaniu pętli). Zapis

$\{P\}$ `while` B `do` I $\{K\}$

oznacza, że dla danych początkowych spełniających warunek P, wykonanie programu W spowoduje, że jeśli tylko program W zakończy swoje działanie, to dane końcowe spełniają warunek K. Naszym celem jest zatem wykazać, że

$\{P\}$ `while` B `do` I $\{K\}$.

Aby zastosować metodę niezmienników należy wymyśleć warunek, nazwijmy go N (niezmiennik), który opisuje własności danych po każdym obrocie pętli (iteracji), a następnie wykazać prawdziwość trzech faktów:

1. $P \Rightarrow N$; Baza indukcji: niezmiennik zachodzi przed wejściem do pętli.
2. $\{N\&B\}\ I\ \{N\}$; Krok indukcyjny: jeśli przed wykonaniem obrotu pętli jest spełniony warunek N oraz jednocześnie jest prawdziwy warunek B, czyli zostanie wykonany kolejny obrót pętli, to po zakończeniu wykonywania instrukcji I warunek N nadal jest spełniony.
3. $\{N\&\neg B\} \Rightarrow K$; Wyjście z pętli: warunek $\neg B$ oznacza, że działanie pętli zostało zakończone. Ponieważ wiemy, że warunek N zachodzi zawsze na początku każdego obrotu pętli, więc zajdzie również wtedy, gdy warunek B po raz pierwszy jest fałszywy. Jeżeli zatem podane wynikanie zachodzi, to jednocześnie mamy zagwarantowane spełnienie warunku K po zakończeniu wykonywania pętli.

Zauważmy, że podana metoda nie gwarantuje tzw. **całkowitej poprawności** programu W, czyli częściowej poprawności wraz z zapewnieniem, że pętla zakończy swoje działanie. Do udowodnienia tego ostatniego faktu (tzw. warunku stopu) stosuje się inne metody.

Wróćmy do naszego zadania. Złożoność czasowa podanego przez nas algorytmu wynosi co nawyżej $n\log_2 n$. Dla każdego z n elementów ciągu danych bowiem, albo dopisujemy go na końcu danych przechowywanych w tablicy `t` (koszt stały), albo w czasie co najwyżej $\log_2 n$ szukamy binarnie odpowiedniego indeksu elementu w tablicy `t`, pod który przypisujemy bieżący element ciągu. Zauważmy, że wyszukiwanie binarne w tym algorytmie nie trwa nigdy dłużej niż $\log_2 10001$, gdyż tablica `t` ma co najwyżej 10001 elementów. Zatem dla dużych wartości n złożoność naszego algorytmu jest ograniczona z góry przez $n\log_2 10001$, czyli jest liniowa względem liczby elementów w ciągu danych wejściowych.

Złożoność pamięciowa algorytmu jest niezależna od długości danych wejściowych i jest równa liczbie różnych dopuszczalnych wartości elementów ciągu, które ograniczają długość najdłuższego podciągu malejącego.

Omówienie rozwiązań podanych przez uczniów

Część uczniów utrudniła sobie to zadanie szukając samego rekordowego podciągu czasów, zamiast tylko jego długości. Zauważmy, że zaproponowana przez nas metoda nie tworzy w tablicy `t` rekordowego podciągu. Gdyby zadanie

brzmiało „wypisz elementy najdłuższego podciągu malejącego", wtedy należałoby zastosować inny algorytm.

Godny odnotowania jest tu następujący algorytm:

1. Posortuj dane od największej do najmniejszej i zapamiętaj je w osobnej tablicy.
2. Rozwiąż zadanie o najdłuższym wspólnym podciągu dla dwóch ciągów: oryginalnego i posortowanego niemalejąco. Sortowanie można wykonać w czasie rzędu $n\log_2 n$, a najdłuższy wspólny podciąg można znaleźć metodą programowania dynamicznego w czasie rzędu n^2, gdzie n jest długością obu ciągów.

W tym algorytmie, poza dłuższym czasem wykonywania mamy jeszcze do czynienia ze sporym obciążeniem pamięci: po pierwsze trzeba zapamiętać dwa ciągi, oryginalny i posortowany, a po drugie – programowanie dynamiczne wymaga użycia dodatkowej tablicy o długości równej długości ciągu wejściowego.

Jeden punkt treści zadania sprawił nieoczekiwany kłopot kilku uczniom. Niby drobiazg, ale kilka osób podało jako odpowiedź długość szukanego ciągu, zamiast jego stokrotnej wartości. Za ten błąd odejmowano kilka punktów, gdyż dla jednego zestawu danych testowych stokrotna wartość długości takiego ciągu nie mieściła się w zakresie typu `Integer`, a za ten błąd z kolei odejmowano punkty tym, którzy tę długość mnożyli przez 100 i powodowali przekroczenie zakresu typu `Integer`.

Kłopot sprawiło również wyjaśnienie podane w treści zadania w nawiasie „(po kropce dziesiętnej)", które w intencji autora było skierowane do tych osób, które w ogóle nie słyszały o przecinku jako znaku oddzielającym (separatorze) część całkowitą od dziesiętnej. Rozumieć je więc należało tak „separatorem będzie przecinek, a jego rola jest taka sama, jak rola kropki dziesiętnej w wielu językach programowania". Tymczasem wielu uczniów zrozumiało to jako dowolność: część całkowita jest oddzielona od części dziesiętnej przecinkiem lub kropką. Nieporozumienia stąd wynikające były rozstrzygane na korzyść uczniów (zostały przygotowane dwa zestawy danych: z przecinkami i kropkami jako separatorami).

Niektórzy uczniowie, podobnie jak w przypadku zadania o trójkątach, zastosowali schemat rozwiązania w postaci:

— wczytaj dane,
— przetwórz dane,
— wypisz wynik.

Podejście to powodowało pojawienie się kłopotów, gdy ciąg danych wejściowych miał długość większą niż wielkość dostępnej pamięci. Jak pokazaliśmy, zadanie można było rozwiązać pobierając elementy jeden po drugim i rozpatrując je na bieżąco bez pamiętania wszystkich dotychczas przeczytanych.

Podany przez nas algorytm nie wymaga wcale większej pamięci dla dłuższych plików wejściowych.

Sporo uczniów nadesłało błędne rozwiązania z powodu błędnych przesłanek, na których oparli swoje algorytmy. Do najbardziej typowych błędów należało założenie, że najdłuższy malejący podciąg można skonstruować używając ciągu wyznaczonego w następujący sposób:

Dla każdego wyrazu a_i ciągu danych wejściowych określić najbliższy następujący po nim element :

— wariant A: od niego mniejszy,
— wariant B: mniejszy od niego o możliwie małą wartość,

a następnie znaleźć rozwiązanie przedłużając ciąg elementami znalezionymi w wariancie A lub B lub w ich kombinacji. Następujący ciąg jest kontrprzykładem dla tych pomysłów:

6 2 4 3 5

Dla elementu 6 najbliższym elementem mniejszym od niego jest 2, a mniejszym elementem możliwie mało różniącym się od 6 jest 5, zaś najdłuższy malejący podciąg 6 4 3 nie zawiera ani 2 ani 5.

Sporo uczniów stosując algorytm opisany w naszym rozwiązaniu nie użyło algorytmu binarnego poszukiwania do znalezienia indeksu w tablicy `t` i szukali go stosując algorytm liniowy, czyli przeglądając elementy tablicy `t` element po elemencie, pogarszając tym koszt całego algorytmu do rzędu n^2 lub $10000n$ dla większych n.

Testy

Testowe dane wejściowe, dla których uruchamiano programy uczniów były czterech rodzajów:

1. Testy podstawowe sprawdzające poprawność algorytmu dla prostych danych.

Test 1: Badanie, czy algorytm nie szuka ciągu nierosnącego, zamiast malejącego. W teście tym powtórzono jedną z liczb – w pozostałych testach ciągi były różnowartościowe, aby nie odejmować punktów dwukrotnie za ten sam błąd.

Test 2: Sprawdzanie, czy nie założono, że w ciągu wynikowym musi wystąpić pierwszy lub ostatni element danych – jest to łatwy do popełnienia błąd przy inicjalizacji niektórych algorytmów. W teście tym ani pierwszy ani ostatni element nie należały do szukanego podciągu.

Test 3: Ciąg danych wejściowych o długości 1.

Test 4: Wyeliminowanie rozwiązań zachłannych, które próbują wydłużać tworzony ciąg przez dodawanie kolejnych elementów wydłużających go.

2. Testy wychwytujące algorytmy o dużej złożoności. Wszystkie testy zawierały dużą ilość danych.

Test 5: Długi ciąg o wykładniczo wielu podciągach malejących i o długości zbliżonej do maksymalnego. Wszystkie algorytmy o pesymistycznej złożoności wykładniczej działały tu wykładniczo długo i były eliminowane ze względu na przekroczenie limitu czasu.

Test 6: Co drugi element ciągu danych należał do maksymalnego podciągu malejącego. Dla $n < 10000$, algorytmy o czasie pesymistycznym rzędu $O(n \log_2 n)$ powinny były wykazać swoją wyższość na tym teście nad algorytmami o złożoności $O(n^2)$. W rzeczywistości przez ten test prześliznęło się kilka algorytmów kwadratowych, wyjątkowo efektywnie zrealizowanych.

3. Testy implementacji, sprawdzające czy użyto odpowiednich typów danych.

Test 7: Ciąg niemalejący zawierający 30000 elementów. Programy, których autorzy zauważyli, że w rozwiązaniu wystarczy użyć tablicy o rozmiarze co najwyżej 10000 (tyle co najwyżej mogło być różnych wyników uzyskanych przez Daniela), nie miały kłopotów z pamięcią, a pozostałe gubiły się próbując jednocześnie pamiętać 30000 liczb.

Test 8: Większa i bardziej złośliwa wersja testu 4.

4. Testy wejścia/wyjścia, badające zdolność programu do odpowiedniego reagowania w sytuacjach błędnych.

Testy 9 i 10: Zły format liczb i liczby spoza zakresu.

Test 11: Litery zamiast cyfr w danych.

7.1.3. Zadanie PIONKI (Autor: Andrzej Walat)

Treść zadania PIONKI

Mamy kwadratową tabliczkę mającą 8×8 pól ponumerowanych jak na rysunku niżej oraz 64 pionki białe i 64 pionki czarne.

Należy dowolne zadane ustawienie pionków na tabliczce – takie, że na każdym polu stoi dokładnie jeden pionek dowolnego koloru – zamienić w możliwie najmniejszej liczbie ruchów na układ jednokolorowy, tzn. taki, że wszystkie pionki na tabliczce mają ten sam kolor.

W pojedynczym ruchu wskazuje się numer dowolnego pola tabliczki – nazwiemy to pole centrum zmian – i zmienia pionki na wszystkich polach przylegających bokiem lub rogiem do wskazanego centrum zmian na pionki o kolorach przeciwnych. Pionka w centrum nie zmienia się. W następnym ruchu wskazuje się kolejne pole jako centrum zmian, odpowiednio zmienia pionki na przyległych polach itd.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Zestaw danych wejściowych jest ciągiem ośmiu słów, zapisanych w ośmiu kolejnych wierszach, odpowiadających ośmiu wierszom tabliczki. Każde słowo składa się z ośmiu liter, B lub C, określających kolory pionków w ośmiu kolejnych polach odpowiedniego wiersza tabliczki.

Odpowiedź powinna zawierać w pierwszym wierszu minimalną liczbę ruchów, jakie należy wykonać. Jeśli ta liczba jest zerem, ten jeden wiersz kończy odpowiedź. W przeciwnym przypadku od nowego wiersza powinny być podane numery kolejnych centrów zmian, oddzielane odstępem lub pojedynczym znakiem końca wiersza.

Gdy jest wiele sposobów otrzymania układu jednobarwnego w minimalnej liczbie ruchów, należy podać tylko jeden z nich.

Jeśli dane są niepoprawne, to znaczy nie spełniają podanych wyżej warunków, odpowiedzią jest słowo NONSENS.

Przykłady

Dla zestawu danych:

```
CCCCCCCC
BBBBBBBB
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
```

przykładem poprawnej odpowiedzi jest:

```
26
57 59 62 64 50 55 42 47 33 35 38 40 25 26 28 29 31 32
18 19 22 23 2 3 6 7
```

Dla zestawu danych:

CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC

poprawną odpowiedzią jest:

0

Dla zestawu danych:

CCCCCCCC
CCDDCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC

poprawną odpowiedzią jest:

NONSENS

Zadanie

Ułóż program, który kolejno dla każdego zestawu danych z pliku PIO.IN generuje właściwą odpowiedź i zapisuje ją w pliku PIO.OUT.

Tekst programu w postaci źródłowej powinien być zapisany w pliku o nazwie PIO.???, gdzie zamiast ??? wstawia się ciąg liter właściwy dla użytego języka programowania.

Program w postaci wykonywalnej powinien mieć nazwę PIO.EXE.

Rozwiązanie nr 1 zadania PIONKI

Rozwiązywanie układów równań liniowych w arytmetyce modulo 2

W arytmetyce modulo 2 mamy tylko dwie wartości 0 i 1. Argumentami i wynikami działań arytmetycznych mogą być tylko te dwie wartości. W szczególności:

$0 + 1 = 1 + 0 = 1$ oraz $0 + 0 = 1 + 1 = 0$;

$0 * 0 = 0 * 1 = 1 * 0 = 0$ oraz $1 * 1 = 1$.

Układy równań liniowych w arytmetyce modulo 2 rozwiązujemy metodami podobnymi, jak w zbiorze liczb rzeczywistych. Jedną z nich jest metoda Gaussa-Jordana, która polega na takim przekształcaniu układu równań, aby współczynniki przy niewiadomych (po lewej stronie znaków równości) utworzyły tzw. macierz jednostkową, czyli mającą jedynki na głównej przekątnej, a poza nią same zera. Wtedy kolumna wyrazów wolnych – po prawej stronie znaków równości – stanowi rozwiązanie układu.

Jako przykład rozwiążmy w arytmetyce modulo 2 układ czterech równań z czterema niewiadomymi:

$$\begin{aligned} a + \ \ + c + d &= 1 \\ d &= 1 \\ a + b + c + d &= 0 \\ b + c + d &= 1 \end{aligned}$$

Faktycznie, zamiast równań zawierających literowe symbole niewiadomych i znaki działań możemy przekształcać odpowiednią macierz (tablicę) współczynników liczbowych stojących przed tymi niewiadomymi:

1 0 1 1 1
0 0 0 1 1
1 1 1 1 0
0 1 1 1 1

Najpierw eliminujemy jedynki z pierwszej kolumny układu, oprócz jedynki znajdującej się na głównej przekątnej. W tym celu dodajemy pierwszy wiersz (równanie) do tych wierszy (równań), w których występuje jedynka na pierwszej pozycji – w naszym przykładzie, do wiersza 3. Pamiętajmy, że jest to dodawanie modulo 2 na każdej pozycji wierszy. Otrzymujemy:

1 0 1 1 1
0 0 0 1 1
0 1 0 0 1
0 1 1 1 1

Następnie eliminujemy jedynki z drugiej kolumny (poza główną przekątną). Przedtem jednak trzeba wprowadzić 1 na drugą pozycję w kolumnie 2. W tym celu przestawiamy wiersze drugi i trzeci:

1 0 1 1 1
0 1 0 0 1
0 0 0 1 1
0 1 1 1 1

a następnie dodajemy drugi wiersz do czwartego:

```
1 0 1 1  1
0 1 0 0  1
0 0 0 1  1
0 0 1 1  0
```

Przestawiamy wiersze trzeci i czwarty, aby wprowadzić jedynkę na główną przekątną w trzeciej kolumnie:

```
1 0 1 1  1
0 1 0 0  0
0 0 1 1  0
0 0 0 1  1
```

i dodajemy trzeci wiersz do pierwszego:

```
1 0 0 0  1
0 1 0 0  0
0 0 1 1  0
0 0 0 1  1
```

Na końcu dodajemy czwarty wiersz do trzeciego:

```
1 0 0 0  1
0 1 0 0  0
0 0 1 0  1
0 0 0 1  1
```

Otrzymaliśmy układ, w którym macierz główna – bez kolumny wyrazów wolnych – jest jednostkowa, a tym samym kolumna wyrazów wolnych jest poszukiwanym rozwiązaniem.

Rozwiązaniem przykładowego układu jest więc: $a = 1$, $b = 0$, $c = 1$, $d = 1$.

Opisane powyżej postępowanie można uogólnić w postaci następującego, nieformalnie zapisanego algorytmu rozwiązywania układu n równań z n niewiadomymi w arytmetyce modulo 2, mając daną tablicę współczynników równania A o rozmiarach n × (n+1).

```
dla i od 1 do n wykonuj:
poczatek
 jesli A[i,i]=0 to
  wyszukaj k>i takie, ze A[k,i]=1, a nastepnie
   przestaw w tablicy A wiersz k z wierszem i;
 dla kazdego j od 1 do n wykonuj:
  jesli j≠i oraz A[j,i]=1 to
   do wiersza k dodaj modulo 2 wiersz i
koniec
```


Trzeba dodać, że opisany algorytm generuje rozwiązanie układu tylko wtedy, gdy układ jest jednoznaczny. W przeciwnym przypadku dla pewnego i nie można wykonać polecenia:

```
jeśli A[i,i]=0 to
 wyszukaj k>i takie, że A[k,i]=1
```

ponieważ dla każdego k ≥ i jest A[k,i]=0.

Dla pełnej poprawności algorytmu należałoby dodać w nim jeszcze jedno zdanie:

```
Jeśli poszukiwane k nie istnieje, zakończ
działanie - układ nie jest jednoznaczny.
```

Na koniec określmy, ile operacji dodawania modulo 2 należy wykonać, aby znaleźć rozwiązanie jednoznacznego układu.

Aby wyeliminować jedynki z i-tej kolumny układu, trzeba w najgorszym przypadku dodać i-ty wiersz do n–1 pozostałych wierszy, a więc $(n–1)(n+1)$ dodawań odpowiednich współczynników liczbowych. Tę operację powtarzamy dla i od 1 do n, więc należy się liczyć z koniecznością wykonania: $n(n–1)(n+1)$ dodawań liczb modulo 2. Ponieważ te operacje decydują o czasie rozwiązania zadania, jest on w przybliżeniu proporcjonalny do n^3. Mówimy, że algorytm Gaussa-Jordana rozwiązywania układu n równań liniowych ma złożoność czasową $O(n^3)$.

Znajdowanie ruchów, jakie należy wykonać, aby zmienić kolory pionków na wybranych polach szachownicy na przeciwne, pozostawiając nie zmienione kolory pionków na pozostałych polach

Każda sekwencja ruchów jest jednoznacznie określona przez podanie ciągu numerów pól szachownicy stanowiących centra zmian kolejnych ruchów.

Fakt, czy pionek na określonym polu ostatecznie zmieni kolor po wykonaniu całej sekwencji ruchów, zależy wyłącznie od tego, ile razy sąsiednie pola będą stanowiły centrum zmian ruchów tworzących tę sekwencję. Jeśli pola sąsiednie będą stanowiły centrum zmian parzystą liczbę razy, to kolor pionka na tym polu zmieni się na przeciwny tyle samo razy, więc ostatecznie nie ulegnie zmianie. W przeciwnym przypadku zmieni się na przeciwny.

Tym samym, jeśli ustalone pole występuje w pewnej sekwencji ruchów jako centrum zmian parzystą liczbę razy, to nie ma wpływu na ostateczną zmianę kolorów pionków na szachownicy. Wszystkie ruchy mające centrum zmian w tym polu można zatem wykreślić bez żadnego skutku dla ostatecznego wy-

niku sekwencji ruchów. Z kolei fakt, że pewne pole występuje jako centrum zmian nieparzystą liczbę razy, ma taki sam wpływ na ostateczny wynik sekwencji ruchów, jak gdyby występowało dokładnie raz – wszystkie powtórzenia ruchu o danym centrum można wykreślić.

Dla każdej sekwencji ruchów istnieje zatem sekwencja mająca taki sam wynik, w której każde pole występuje jako centrum zmian co najwyżej jeden raz.

W dalszej części możemy więc rozważać wyłącznie sekwencje ruchów bez powtórzeń.

Dla dowolnej sekwencji ruchów bez powtórzeń S, **sygnaturą** S nazywamy ciąg $[c_i]_{i \in [1..64]}$ złożony z zer i jedynek, taki że:

$$c_i = \begin{cases} 1, & \text{gdy pole o numerze } i \text{ występuje jako centrum zmian} \\ & \text{w ciągu } S, \\ 0, & \text{gdy nie występuje.} \end{cases}$$

Wynikiem wykonania sekwencji ruchów S nazywamy ciąg $[z_i]_{i \in [1..64]}$ spełniający:

$$z_i = \begin{cases} 1, & \text{gdy pole o numerze } i \text{ zmienia kolor po wykonaniu ciągu } S, \\ 0, & \text{w przeciwnym przypadku.} \end{cases}$$

Łatwo zauważyć, że dowolne dwie sekwencje ruchów bez powtórzeń o takiej samej sygnaturze mają taki sam wynik oraz że pomiędzy sygnaturą sekwencji ruchów a jej wynikiem zachodzi zależność mająca postać układu 64 równości:

$$\begin{aligned}
&(1) && c_2 + c_9 + c_{10} = z_1 \\
&(2) && c_1 + c_3 + c_9 + c_{10} + c_{11} = z_2 \\
&(3) && c_2 + c_4 + c_{10} + c_{11} + c_{12} = z_3 \\
&\quad && \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&(35) && c_{26} + c_{27} + c_{28} + c_{34} + c_{36} + c_{42} + c_{43} + c_{44} = z_{35} \\
&\quad && \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&(64) && c_{55} + c_{56} + c_{63} = z_{64}
\end{aligned}$$

w których plus jest znakiem dodawania modulo 2. Na przykład, pole numer 1 zmienia kolor tylko wtedy, gdy wszystkie trzy lub dokładnie jedno z trzech sąsiednich pól wystąpi jako centrum zmian.

Jeśli dany jest wynik zmian w postaci ciągu $[z_i]$, to znalezienie odpowiedniej sekwencji ruchów, która daje taki wynik, sprowadza się do rozwiązania powyższego układu równań liniowych z 64 niewiadomymi $[c_i]$ w arytmetyce modulo 2 dla danych $[z_i]$.

Jest to – dla dowolnego ciągu danych $[z_i]$ – układ jednoznaczny; decyduje o tym macierz główna układu utworzona przez współczynniki przy niewiadomych. Można sprawdzić, że gdyby szachownica miała rozmiar 5 na 5, to odpowiedni układ 25 równań nie byłby jednoznaczny.

Jako centra zmian kolejnych ruchów powodujących pożądane zmiany, należy wybrać – w dowolnej kolejności – pola o numerach i, dla których $c_i = 1$.

Algorytm sprowadzania dowolnej konfiguracji pionków do jednokolorowej

1. Znajdź ruchy, jakie należy wykonać, aby zamienić wszystkie pionki na kolor biały. W tym celu przyjmij, że należy zmienić kolor wszystkich pionków czarnych i rozwiąż układ równań liniowych dla odpowiedniego ciągu $[z_i]$.
2. Ustal inny ciąg danych $[z_i]$ przyjmując, że należy zmienić kolor wszystkich pionków białych i rozwiąż odpowiedni układ równań liniowych. W ten sposób znajdowane są ruchy, jakie należy wykonać, aby zamienić pionki na wszystkich polach na kolor czarny.
3. Sprawdź, w którym z dwóch powyższych przypadków liczba wymaganych ruchów jest mniejsza i przyjmij tę liczbę oraz odpowiednią sekwencję ruchów za rozwiązanie zadania.

Ocena złożoności czasowej rozwiązania nr 1

Uzasadniliśmy powyżej, że czas rozwiązania układu n równań liniowych metodą Gaussa-Jordana jest proporcjonalny do n^3.

W naszym zadaniu, rozmiar układu (czyli rozmiar tablicy), a tym samym rozmiar danych jest ustalony. Każdy zestaw danych składa się z ośmiu wierszy po osiem znaków B lub C, czyli $n = 64$.

Gdybyśmy jednak przyjęli, że tablica ma dowolny rozmiar $m > 5$, to wtedy czas obliczeń byłby proporcjonalny do m^6, gdyż decyduje o nim czas rozwiązania dwóch układów równań z m^2 niewiadomymi.

W takim przypadku należałoby się zastanowić również nad sposobem reprezentowania danych, aby nie przekroczyć pojemności pamięci.

Chociaż rozmiar zadania jest ustalony, analiza złożoności czasowej jest istotna, ponieważ wyjaśnia, czy rozwiązanie można znaleźć w rozsądnym czasie.

Rozwiązanie nr 2 zadania PIONKI

Metoda szybkiego wyznaczania ruchów, jakie trzeba wykonać, aby zmienić kolory pionków na wybranych polach

Dla każdego i od 1 do 64, znajdujemy (dowolną metodą – na przykład opisaną w rozwiązaniu 1) ruchy, jakie należy wykonać, aby zmienić kolor pionka na polu i, nie zmieniając kolorów pionków na pozostałych polach.

Każde z tych 64 rozwiązań szczególnych przypadków można zapisać zgodnie z przyjętą powyżej konwencją notacyjną w postaci odpowiedniego wektora złożonego z 64 zer i jedynek:

$$\boldsymbol{c}_i = [\,c_{ij}\,]_{j \in [1..64]}$$

Na przykład, aby zmienić kolor pionka na jednym tylko polu o numerze 1, trzeba wykonać co najmniej 32 ruchy, wybierając jako centra zmian pola: 2, 4, 6, 8, 9, 10, 12, 13, 15, 16, 22, 25, 26, 28, 29, 30, 34, 36, 41, 43, 44, 46, 47, 48, 50, 54, 56, 57, 58, 62, 63, 64.

To rozwiązanie można zapisać w postaci wektora

$\boldsymbol{c}_1$ = [01010101111011011000001001101110001010000101101110100010111000111].

Rozwiązaniem zadania dla przypadku $i = 2$ jest wektor

$\boldsymbol{c}_2$ = [10000000110110110000101011010110100010000011010110101000111011101].

Zapisując te wektory jeden pod drugim, utworzymy macierz kwadratową $\boldsymbol{C}$ złożoną z 64 wierszy i 64 kolumn (zob. Dodatek na końcu tego punktu).

Rozwiązaniem zadania, jak zmienić kolory pionków na ustalonych polach dla dowolnego ustawienia zapisanego w postaci wektora $\boldsymbol{z} = [\,z_i\,]$, jest suma (modulo 2) odpowiednich wektorów $\boldsymbol{c}_i$ – wierszy macierzy $\boldsymbol{C}$. Ściśle, wektor $\boldsymbol{c}_i$ (i-ty wiersz macierzy) jest składnikiem odpowiedniej sumy $\boldsymbol{r}$ wtedy i tylko wtedy, gdy $z_i = 1$.

Na przykład, rozwiązaniem zadania, jak zmienić kolory pionków na trzech polach: 2, 7 i 9 jest wektor:

$$\boldsymbol{r} = \boldsymbol{c}_2 + \boldsymbol{c}_7 + \boldsymbol{c}_9.$$

Zależność wektora $\boldsymbol{r}$ od danego ustalenia $\boldsymbol{z}$ można zapisać symbolicznie w następujący sposób:

$$\boldsymbol{r} = [\,r_i\,], \qquad r_i = \sum_{j=1..64} c_{ij} z_j.$$

W algebrze tę operację, która dla danej macierzy $\boldsymbol{C}$ oraz wektora $\boldsymbol{z}$ daje jako wynik wektor $\boldsymbol{r}$, nazywamy mnożeniem macierzy $\boldsymbol{C}$ przez wektor $\boldsymbol{z}$.

Procedura wyznaczania ruchów, aby zmienić ustalone pola, polega więc na obliczaniu iloczynu stałej macierzy $\boldsymbol{C}$ przez dowolny wektor $\boldsymbol{z}$ stanowiący parametr procedury:

```
procedure WyznaczRuchy(var z,r:WektorMBitowy);
 var i,j:Byte;
begin
 for i:=1 to M do begin
  r[i]:=0;
  for j:=1 to M do
   r[i]:=(r[i]+c[i,j]*z[j]) mod 2
 end
end; {WyznaczRuchy}
```


Doprowadzanie dowolnej konfiguracji pionków do jednokolorowej

To rozwiązanie zadania redukuje się do dwukrotnego wywołania procedury mnożenia macierzy przez wektor, dla znalezienia ruchów doprowadzających dany układ pionków do jednobarwnego: białego i czarnego oraz wybrania rozwiązania lepszego, tzn. takiego, które doprowadza do układu jednobarwnego w mniejszej liczbie ruchów.

Ocena złożoności czasowej rozwiązania nr 2

W przypadku tego rozwiązania, decydujący wpływ na czas znalezienia rozwiązania ma dwukrotne mnożenie macierzy kwadratowej o rozmiarze 64 (m^2) przez wektor. Czas wykonania zadania jest w przybliżeniu proporcjonalny do m^4, ponieważ mnożenie macierzy mającej n wierszy i kolumn przez wektor mający n współrzędnych wymaga wykonania n^2 mnożeń i dodawań.

Ten sposób rozwiązania zadania ma złożoność czasową $O(m^4)$ i jest istotnie szybszy niż rozwiązanie nr 1.

Omówienie rozwiązań podanych przez uczniów

Spośród 59 uczniów, którzy przysłali całkowicie poprawne rozwiązania tego zadania i otrzymali za nie maksymalną liczbę 100 punktów, niektórzy wybrali metodę polegającą na rozwiązaniu układów 64 równań liniowych, inni metodę opisaną powyżej jako rozwiązanie nr 2.

W pierwszym przypadku, niektórzy uczniowie cytowali podręcznikowe nazwy wybranych metod, np. Gaussa-Jordana.

W drugim przypadku, z reguły opisywali wynalezioną przez siebie metodę własnymi słowami, bez odwoływania się do języka algebry, być może nie mając świadomości, że definiowana przez nich operacja „wyznaczania rozwiązania na podstawie matryc zapisanych jako stałe programu" – jest opisana w literaturze w nieco bardziej sformalizowanej postaci jako mnożenie macierzy.

Te rozwiązania dawały wynik najszybciej, w ciągu ułamka sekundy, a programy były stosunkowo krótkie i zrozumiałe.

Inne metody polegały na jakiejś formie badania, jakie ruchy mogą przynieść pożądany skutek. Różniły się one znacznie pod względem efektywności eliminowania dróg, które nie mogą doprowadzić do wyniku i ograniczenia liczby

rozpatrywanych wariantów. Najszybsze z nich znajdowały wynik dla dowolnego ustawienia początkowego pionków w czasie kilkudziesięciu sekund i mieściły się w ustalonych limitach czasowych.

Metody siłowe, oparte na badaniu krok po kroku konsekwencji wszystkich możliwych ruchów, prowadziły do wyniku tylko w skrajnie prostych przypadkach, kiedy układ początkowy był jednokolorowy lub mógł być doprowadzony do takiego stanu w małej liczbie ruchów. Ponieważ takie testy były punktowane najniżej, rozwiązania te zostały ocenione stosunkowo nisko.

Niektórzy zawodnicy popełniali błąd, polegający na przyjęciu fałszywego założenia, że jeśli liczba pionków białych na szachownicy jest większa niż liczba pionków czarnych (i odpowiednio odwrotnie), to doprowadzenie układu do jednokolorowej konfiguracji białej wymaga mniejszej liczby ruchów niż doprowadzenie go do konfiguracji czarnej. W rezultacie w pewnych przypadkach ich program znajdował ruchy doprowadzające do układu jednobarwnego, ale liczba ruchów nie była minimalna. Za taki wynik testu przydzielano tylko część maksymalnej liczby punktów.

Testy

W pierwszych dwóch testach dane były plansze jednobarwne: biała i czarna, w trzecim i czwartym, układ jednobarwny można było otrzymać po wykonaniu odpowiednio jednego oraz trzech ruchów. Dla takich danych nawet nieoptymalne, ale poprawne programy znajdowały rozwiązania w krótkim czasie.

Testem 5 był pierwszy przykład z treści zadania. W tym przypadku należy wykonać 26 ruchów i chociaż w sytuacji początkowej na planszy jest 56 pionków czarnych, a tylko 8 białych, doprowadzenie do układu jednobarwnego białego wymaga mniejszej liczby ruchów, niż do czarnego.

Testy 6 i 7 były przykładami, w których rozwiązanie składa się z wielu ruchów, odpowiednio 25 i 24:

CCCCCCCC	CBCBCBCB
BBBBBBBB	BCBCBCBC
CCCCCCCC	CBCBCBCB
CBBBBBCC	BCBCBCBC
CCCCCCCC	CBCBCBCB
CCCBCCCC	BCBCBCBC
CCCCCBCC	CBCBCBCB
CCCCCCCC	BCBCBCBC

Kolejne testy – od 7 do 11 – miały na celu sprawdzenie, czy program wykrywa błędne dane i zgodnie z warunkami zadania wpisuje do pliku wyników słowo NONSENS.

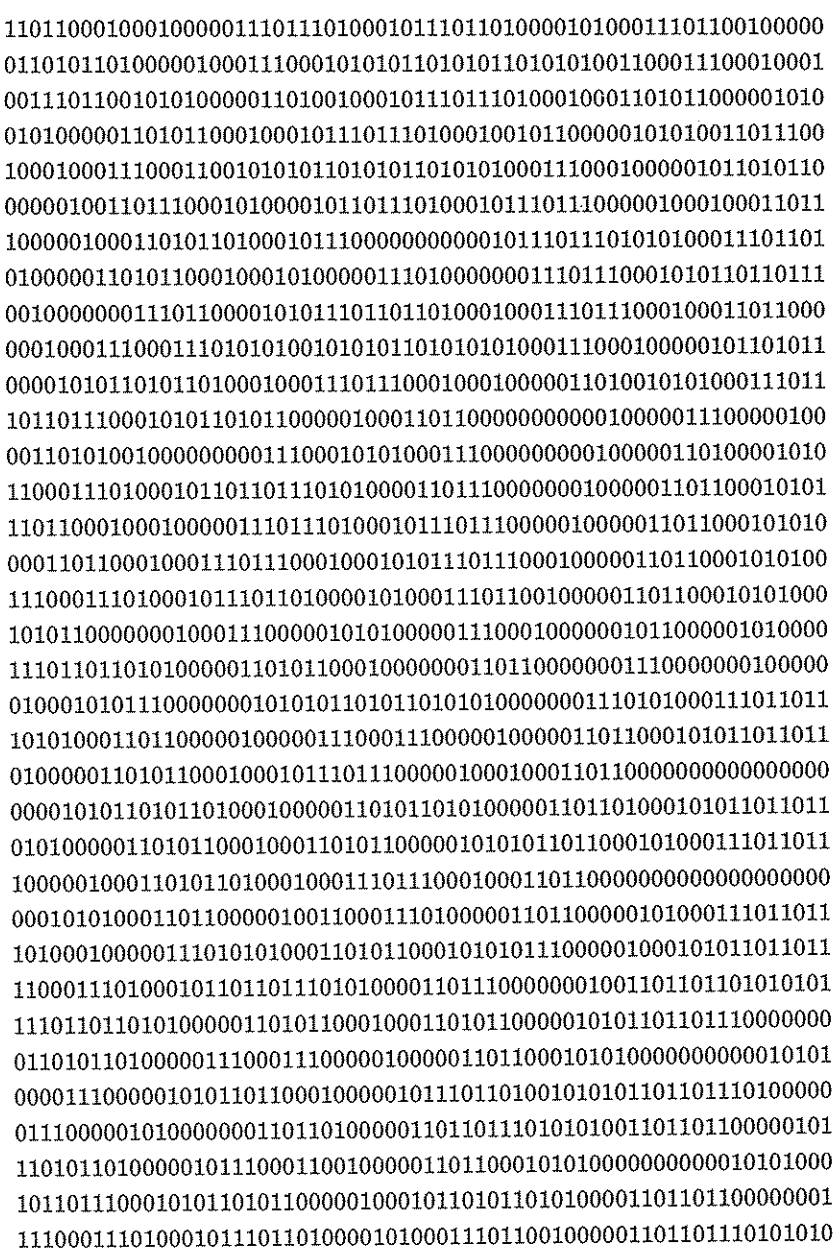
W każdym z tych testów wystąpił któryś z błędów w danych:

- za dużo wierszy w jednym zestawie danych,
- za mało wierszy w jednym zestawie danych,
- za dużo znaków w jednym wierszu zestawu danych,
- za mało znaków w jednym wierszu zestawu danych,
- niedopuszczalny znak.

Miały one mniejszą wagę – za poprawne przejście można było otrzymać 15 (z ogólnej liczby 100) punktów.

Dodatek – macierz C

0101010111011011000001001101110001010000101101110100010111000111
1000000011011011000010101101011010001000001101011010100011101101
0001010100000000000101010001101100000100110001110100000101101011
1010000011011011001010101110110110000010110110000000101000001110
0000010111011011010101001011011101000001000110110101000001110000
1010100000000000101010001101100000100000111000111000001011010110
0000000111011011010100000110101100010001101011000001010110110111
1010101011011011001000000011101100001010111011011010001011100011
1101101101010001000001110101010001101011000101010111000001000101
1101101110001010000011011000001011100011001000001101100010101000
0000000000000000000011011000100011101110001000101101011000010000 1
1101101110001010001101101010100000110101100010001101011000001010
1101101101010001011011000001010110101100000100010110101101010000
0000000000000000110110001000100000111011101000100011010110000010
1101101101010001101100000100000111000111000001000001101100010101
1101101110001010111000000010101011010110100000101000000111010100010
0000010000000111000000011011000000010001101011000001010110110111
0000101000001101000000100011100000101010000011100010000000110101
0001010100011011000001000111011101010001000111011100010001101 1000
0010101000110110000010001110111010100010001110111000100011011000
0101010001101100000100000111011101000101110111000001000100011011
1010100011011000001000000011101100001010111011011010001011100011
0101000010110000010000000001110001010100011100000000010010101100
0010000011100000100000000000110110001000001101011010100011101101
1101110001010100101100000100010001110111000100010110101101010000
1101011010000010001110001010101011010101001010101110001110001000
0001101100010001110111100010001011011011101010000110111000000100
1110110110101000111011100000001011100000101000100011010110000010
1011011100010101011101110100000000000111010001011010110001000001



7.2. Zawody II stopnia

7.2.1. Zadanie SUMA KWADRATÓW CYFR (Autor: Wojciech Rytter)

W dalszej treści używamy skróconej nazwy tego zadania SUMA.

Treść zadania SUMA

W zbiorze *100 zadań* sławnego polskiego matematyka Hugona Steinhausa* znajduje się następujący problem:

Udowodnij, że jeśli dowolną dodatnią liczbę naturalną n zastąpimy sumą kwadratów jej cyfr, a następnie otrzymaną w ten sposób nową liczbę sumą kwadratów jej cyfr i tak dalej, to po pewnej skończonej liczbie powtórzeń tej operacji otrzymamy liczbę 1 albo 4.

Przykład

suma kwadratów cyfr liczby 89 wynosi 64 + 81 = 145,
suma kwadratów cyfr liczby 145 wynosi 1 + 16 + 25 = 42,
suma kwadratów cyfr liczby 42 wynosi 16 + 4 = 20,
suma kwadratów cyfr liczby 20 wynosi 4.

Zadanie

1. Uzasadnij, że nieformalnie zapisany algorytm:

   `dopóki` $n \notin \{1,4\}$ `wykonuj` `n:=SumaKwadratówCyfr(n)`

   ma własność stopu, to znaczy dla każdej wartości naturalnej n zatrzymuje się po pewnym skończonym czasie.

   Wykorzystaj w dowodzie komputer, ale oczywiście nie możesz przetestować wszystkich liczb naturalnych. Napisane na kartce uzasadnienie stanowi istotną część rozwiązania zadania.
2. Ułóż program, który kolejno dla każdej liczby naturalnej dodatniej n z pliku tekstowego KWA.IN generuje i zapisuje w pliku tekstowym KWA.OUT ciąg różnych liczb, zaczynający się od n i kończący liczbą 1 albo 4, w którym każda kolejna liczba, prócz pierwszej, jest sumą kwadratów cyfr poprzedniej liczby w tym ciągu. Program powinien wykonywać zadanie nawet dla bardzo dużych liczb naturalnych $n < 10^{66}$.

   Na początku programu podaj w komentarzu Twoje nazwisko, imię i numer stanowiska, na którym pracujesz.

---

* H. Steinhaus, *100 zadań*, PHU „DIP", Warszawa 1993.

Plik wejściowy KWA.IN jest zawsze niepusty i zawiera dodatnie liczby naturalne w postaci dziesiętnej (co najwyżej 66-cyfrowe). Pomiędzy kolejnymi liczbami jest odstęp lub jeden znak końca wiersza, a bezpośrednio po ostatniej liczbie w pliku jest znak końca pliku. Dane są zapisane bezbłędnie i Twój program nie musi sprawdzać ich poprawności.

Plik wyjściowy KWA.OUT powinien zawierać ciągi generowane dla liczb pobieranych z pliku KWA.IN. Pomiędzy kolejnymi liczbami każdego ciągu jest odstęp lub jeden znak końca wiersza. Kolejne dwa ciągi powinien oddzielać pusty wiersz, bezpośrednio po ostatnim wyrazie ostatniego ciągu powinien wystąpić znak końca pliku. Poza tym nie powinno być w nim żadnych innych zapisów.

Przykłady

Dla pliku KWA.IN:

33 17 638 *(znak końca pliku)*

plik KWA.OUT powinien zawierać ciągi:

33 18 65 61 37 58 89 145 42 20 4
(pusty wiersz)
17 50 25 29 85 89 145 42 20 4
(pusty wiersz)
638 109 82 68 100 1 *(znak końca pliku)*

Dla pliku KWA.IN:

89 *(znak końca pliku)*

plik KWA.OUT powinien zawierać jeden ciąg:

89 145 42 20 4 *(znak końca pliku)*

Twój program powinien szukać pliku KWA.IN w katalogu bieżącym i tworzyć plik KWA.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę KWA.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku KWA.EXE. Oba te pliki powinny być zapisane na dysku stałym oraz na dyskietce.

Rozwiązanie zadania składa się z programu – tylko jednego – w postaci źródłowej i wykonywalnej, zapisanego zgodnie z podanymi wyżej warunkami na dysku stałym i dyskietce oraz pisemnego uzasadnienia, że nieformalny algorytm sformułowany w punkcie 1. zadania ma własność stopu.

Rozwiązanie zadania SUMA

Rozwiązanie zadania składa się z dwóch części:

— dowodu własności sumy kwadratów cyfr dla liczb większych od pewnej liczby m oraz
— komputerowego sprawdzenia prawdziwości tezy z treści zadania dla liczb mniejszych od m.

W teorii liczb jest wiele twierdzeń, których dowód poprawności przebiega w podobny sposób.

Omówmy najpierw program wykonujący tę drugą część rozwiązania.

Ponieważ liczby w danych wejściowych mogą mieć aż 66 cyfr przyjmujemy, że w naszym programie liczby są reprezentowane w postaci łańcucha cyfr, czyli są typu `string`. Program generuje ciąg liczb, z których każda jest sumą kwadratów cyfr poprzedniej liczby. W tym celu korzystamy z funkcji `SumaKwadratowCyfr`, której wartością jest suma kwadratów cyfr jej argumentu. Można ją zapisać następująco:

```
function SumaKwadratowCyfr(Liczba:string):string;
 var SumaKwadr :Word;
     Cyfra,Poz :Byte;
     NowaLiczba:string;
begin
 SumaKwadr:=0;
 for Poz:=1 to Length(Liczba) do begin
  Cyfra:=Ord(Liczba[Poz])-48;
  Cyfra:=Cyfra*Cyfra;
  Inc(SumaKwadr,Cyfra)
 end;
 Str(SumaKwadr,NowaLiczba);
 SumaKwadratowCyfr:=NowaLiczba
end; {SumaKwadratowCyfr}
```

Generowanie ciągu liczb może być zapisane rekurencyjnie lub iteracyjnie – podamy zapis rekurencyjny:

```
procedure GenerowanieCiagu(Liczba:string);
begin
 Write(KWA.OUT,Liczba);  Write(KWA.OUT, ' ');
 if not ((Liczba='1') or (Liczba='4')) then
  GenerowanieCiagu(SumaKwadratowCyfr(Liczba))
end; {GenerowanieCiagu}
```

Główny program jest bardzo prosty. Inicjalizujemy plik wejściowy KWA.IN oraz plik wyjściowy KWA.OUT a następnie, dopóki plik wejściowy zawiera przynajmniej jeden element, wczytujemy kolejną liczbę `NowaLiczba` jako łańcuch i wywołujemy procedurę `GenerowanieCiagu(NowaLiczba)`.

Dowód tego, że procedura `GenerowanieCiagu` ma własność stopu składa się z części: matematycznej (teoretycznej) i komputerowej. W części matematycznej dowodzimy prawdziwości faktu, że dla $m = 100$, jeśli $n \geq m$, czyli jeśli n ma co najmniej 3 cyfry, to $\mathrm{skc}(n) < n$, gdzie $\mathrm{skc}(n)$ oznacza sumę kwadratów cyfr liczby n, traktowaną jako liczbę. Zauważmy, że wartością funkcji `SumaKwadratowCyfr` jest reprezentacja łańcuchowa wartości funkcji $\mathrm{skc}(n)$.

Twierdzenie. *Jeśli* $n \geq 100$, *to* $\mathrm{skc}(n) < n$.

Dowód. Niech $n = a_l 10^l + a_{l-1} 10^{l-1} + \ldots + a_1 10 + a_0$. Wtedy $\mathrm{skc}(n) = a_l^2 + \ldots + a_0^2$. Łatwo zauważyć, że

$$n - \mathrm{skc}(n) \geq a_l (10^l - a_l) - a_0 (a_0 - 1).$$

W wyrażeniu po prawej stronie mamy różnicę dwóch liczb. Dla $l \geq 3$ minimalną wartością liczby $a_l (10^l - a_l)$ jest (100–1). Natomiast maksymalną wartością liczby $a_0 (a_0 - 1)$ jest 9(9–1) = 72. Zatem różnica ta jest dodatnia dla $l \geq 3$ – udowodniliśmy więc nasze twierdzenie.

Na podstawie tego twierdzenia wiemy, że jeśli dana jest liczba nie mniejsza niż 100, to po pewnym czasie w ciągu liczb będących sumą kwadratów cyfr poprzedniej liczby pojawi się liczba mniejsza niż 100. Pozostaje zatem i wystarczy wykonać obliczenia opisanym programem dla wszystkich liczb dwucyfrowych.

Druga część dowodu polega na sprawdzeniu prawdziwości faktu z treści zadania przez program komputerowy dla wszystkich liczb mniejszych od 100. Liczb tych jest niewiele i wywołanie dla każdej z nich procedury `GenerowanieCiagu` nie zajmie dużo czasu. Tym niemniej można to jeszcze przyspieszyć zapamiętując w pomocniczej tablicy sprawdzone już liczby i jeśli w ciągu generowanym dla danej liczby otrzymamy wartość odznaczoną już w tej tablicy, to możemy przerwać dalsze generowanie.

Omówienie rozwiązań podanych przez uczniów

Zadanie było bardzo proste i nie sprawiło uczniom większych trudności. Wielu uczniów jednak zamiast dla liczb mniejszych od 100 wykonywało obliczenia na komputerze dla wszystkich liczby mniejszych od 1000 – najwyraźniej nie wiedzieli o fakcie wymienionym w twierdzeniu powyżej. Niektórzy zauważyli, że pomimo tego, iż początkowe liczby mogą mieć 66 cyfr, sumy kwadratów ich cyfr

nie wykraczają swoją wartością poza zakres zmiennej typu `Word`, a dokładniej nie przekraczają liczby 66*81 = 5346. Można by zatem, zmienić typ funkcji `SumaKwadratowCyfr` ze `string` na `Word`.

Testy

Testy składały się z liczb bardzo długich (np. utworzonych z 63 i 50 dziewiątek), liczb podanych w treści zadania oraz z ciągu liczb od 1 do 19. Te pierwsze dane liczby miały na celu sprawdzenie, czy programy przyjmują długie liczby, jako dane wejściowe.

Najdłuższy ciąg liczb, utworzony z sum kwadratów liczb poprzednich otrzymano dla liczby 6 – składa się on z 14 liczb: 6 36 45 41 17 50 25 29 85 89 145 42 20 4.

7.2.2. Zadanie PRZEDSIĘWZIĘCIE (Autor: Maciej M. Sysło)

Treść zadania PRZEDSIĘWZIĘCIE

Przedsięwzięcie składa się z pewnej liczby czynności ponumerowanych kolejnymi liczbami naturalnymi od 1 do $N \leq 1000$. Przedsięwzięcie będzie wykonane, gdy zostaną wykonane wszystkie czynności.

Dla każdej czynności jest znany czas jej wykonania, będący liczbą naturalną dodatnią oraz zbiór tych czynności, które muszą zostać wykonane przed jej rozpoczęciem.

Przedsięwzięcie jest **wykonalne**, jeśli nie zawiera żadnego podzbioru czynności tworzących **cykl**. Na przykład, jeśli czynność a musi zostać wykonana przed rozpoczęciem czynności b, czynność b musi zostać wykonana przed rozpoczęciem czynności c, czynność c musi zostać wykonana przed rozpoczęciem czynności d oraz czynność d musi zostać wykonana przed rozpoczęciem czynności a, to czynności te tworzą cykl (a, b, c, d) i przedsięwzięcie nie jest wykonalne, bo żadnej z czynności należących do tego cyklu nie będzie można rozpocząć.

Zadanie

Ułóż program, który kolejno dla każdego zestawu danych o przedsięwzięciu, wczytanego z pliku tekstowego PRZ.IN:

1. stwierdza, czy przedsięwzięcie jest wykonalne,
2. wyznacza najkrótszy czas wykonania przedsięwzięcia,

3. odpowiada na zapytania, czy przedłużenie wykonywania podanej czynności o podany czas wydłuży czas wykonania całego przedsięwzięcia

i zapisuje wyniki w pliku tekstowym PRZ.OUT.

Na początku programu podaj w komentarzu Twoje nazwisko i imię i numer stanowiska, na którym pracujesz.

Plik PRZ.IN jest zawsze niepusty i może zawierać wiele zestawów danych o różnych przedsięwzięciach.

Każdy zestaw danych o przedsięwzięciu ma następującą postać:

W pierwszym wierszu jest zapisana liczba czynności N, z których składa się przedsięwzięcie.

Po nim występuje N opisów czynności.

Każdy opis czynności jest skończonym co najmniej dwuwyrazowym ciągiem dodatnich liczb naturalnych oddzielonych odstępem lub znakiem końca wiersza. Pierwsza liczba w ciągu jest numerem czynności, druga – czasem jej wykonywania, a kolejne liczby są numerami czynności, które muszą być wykonane wcześniej. Opis czynności jest zakończony średnikiem, po którym występuje koniec wiersza.

Po ostatnim opisie czynności może nastąpić ciąg zapytań, czy wydłużenie czynności o podanym numerze o określony czas spowoduje opóźnienie realizacji przedsięwzięcia.

Każde zapytanie ma postać pary liczb, oddzielonych odstępem i jest zakończone średnikiem, po którym może wystąpić odstęp lub koniec wiersza. Pierwsza liczba jest numerem czynności, a druga – proponowanym wydłużeniem czasu jej wykonania.

Kolejne zestawy danych w pliku PRZ.IN są oddzielone pustym wierszem. Po ostatnim zestawie danych następuje koniec pliku.

Zakładamy, że dane w pliku PRZ.IN zostały zapisane bezbłędnie i Twój program nie musi sprawdzać ich poprawności.

Plik PRZ.OUT powinien zawierać po jednym zestawie wyników dla każdego zestawu danych o przedsięwzięciu z pliku PRZ.IN.

Zestawem wyników jest:

— słowo CYKL, jeśli przedsięwzięcie jest niewykonalne, albo
— liczba naturalna (najkrótszy czas wykonania przedsięwzięcia), gdy zestaw danych nie zawiera zapytań, albo
— liczba naturalna oraz słowa TAK lub NIE – wpisane w oddzielnych wierszach – które są odpowiedziami na kolejne zapytania.

Kolejne zestawy wyników są oddzielane pustym wierszem, po ostatnim zestawie występuje koniec pliku.

Poza tym w pliku PRZ.OUT nie powinno być żadnych innych zapisów.

Przykład

Dla pliku PRZ.IN:

```
6
1 1;
2 1 1 6;
5 4 1;
3 5 5 4;
4 5;
6 2 4;
4 1; 2 1; 6 2; 1 1; 2 3; 6 3;
(pusty wiersz)
4
1 1 3;
2 1 1;
3 2 1 2;
4 1 3;
2 1; 3 2;
(pusty wiersz)
4
1 1;
2 2;
3 3 1;
4 2 3;
1 1; 2 3; 2 5; (koniec pliku)
```

Twój program powinien utworzyć następujący plik PRZ.OUT:

```
10
TAK
NIE
NIE
TAK
TAK
TAK
(pusty wiersz)
CYKL
(pusty wiersz)
6
TAK
NIE
TAK (koniec pliku)
```

Na początku programu podaj w komentarzu Twoje nazwisko, imię i numer stanowiska, na którym pracujesz.

Twój program powinien szukać pliku PRZ.IN w katalogu bieżącym i tworzyć plik PRZ.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę PRZ.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku PRZ.EXE. Oba te pliki powinny być zapisane na dysku stałym oraz na dyskietce.

Rozwiązanie zadania składa się z programu – tylko jednego – w postaci źródłowej i wykonywalnej, zapisanego zgodnie z podanymi wyżej warunkami na dysku stałym i dyskietce oraz opisu algorytmu rozwiązania zadania wraz z uzasadnieniem jego poprawności.

Rozwiązanie zadania PRZEDSIĘWZIĘCIE

Zadanie, dla którego mamy podać rozwiązanie jest określone na strukturze danych, którą nazywamy siecią. **Sieć** składa się z **digrafu** (tj. **grafu skierowanego**) i pewnej liczby funkcji opisanych na jego elementach. W naszym przypadku, dane o przedsięwzięciu można zapisać w postaci sieci. Digraf tej sieci tworzy **zbiór wierzchołków** odpowiadający czynnościom przedsięwzięcia i **zbiór łuków** utworzony z uporządkowanych par wierzchołków, odpowiadających czynnościom związanym relacją poprzedzania. W danych występuje jeszcze czas wykonania czynności – w sieci reprezentującej przedsięwzięcie odpowiada mu **funkcja** opisana na zbiorze wierzchołków.

W rozwiązaniu zadania sieć jest reprezentowana przez tablicę rekordów, typu `Digraf`, po jednym dla każdego wierzchołka (czyli dla każdej czynności). Każdy rekord z kolei jest złożony z pól zawierających lokalne informacje związane z czynnością, w tym m.in.: czas jej wykonania, listę czynności bezpośrednio poprzedzających (`WeLista`) i listę czynności bezpośrednio następujących (`WyLista` – tworzymy ją na podstawie tej pierwszej listy) oraz liczebności tych list (`WeStopien` i `WyStopien`, odpowiednio), nazywane **stopniem wejściowym** i **stopniem wyjściowym** wierzchołka. Odpowiednie definicje typów mają następującą postać:

```
const
 Nmax=100;
type
 NumWierzch =1..Nmax;
 Wektor     =array[NumWierzch] of Integer;
```



```
PDigrafElem=^TDigrafElem;
TDigrafElem=record
             Wierzch :NumWierzch;
             Nastepny:PDigrafElem
            end;
Wierzcholek=record
             Czas                :Integer;
             WeLista,WyLista     :PDigrafElem;
             WeStopien,WyStopien:Integer
            end;
Digraf     =array[NumWierzch] of Wierzcholek;
```

Korzystając z tych struktur danych można wczytać dane wejściowe dla naszego zadania – dla pojedynczej czynności o numerze `c` wykonuje to procedura `CzytajOpisCzynnosci` (dla uproszczenia programu, postać danych została nieco zmieniona – zamiast średnika na końcu, ciąg czynności poprzedzających czynność `c` jest poprzedzony liczbą takich czynności `l`). We fragmentach programów dane wejściowe opisujące przedsięwzięcie zostają umieszczone i są reprezentowane w zmiennej globalnej `Dig` typu `Digraf`.

```
procedure CzytajOpisCzynnosci(c:NumWierzch);
 var j,k,l,w:Integer;
     p,q    :PDigrafElem;
begin
 Read(w,l);
 with Dig[c] do begin
  Czas:=w;
  WeStopien:=l;
  for k:=1 to l do begin
   New(p);  Read(j);
   p^.Wierzch:=j;  p^.Nastepny:=WeLista;
   WeLista:=p;
   New(q);  q^.Wierzch:=c;
   with Dig[j] do begin
    WyStopien:=WyStopien+1;
    q^.Nastepny:=WyLista;
    WyLista:=q
   end
  end {for}
 end {with}
end; {CzytajOpisCzynnosci}
```

Mając zdefiniowane struktury danych oraz wczytane dane wejściowe możemy przystąpić do podania algorytmów dla trzech wyróżnionych podzadań:

— określić czy przedsięwzięcie jest wykonalne, tzn. czy nie zawiera czynności układających się w cykl;
— znaleźć najkrótszy czas wykonania całego przedsięwzięcia;
— dla wybranej czynności i przedłużenia czasu jej wykonania, określić czy to przedłużenie może wpłynąć na wydłużenie wykonywania całego przedsięwzięcia.

Rozwiązanie, które podamy ma tę cechę, że algorytm rozwiązywania pierwszego podzadania przygotowuje dodatkowe informacje o sieci dla algorytmu rozwiązywania drugiego podzadania, i z kolei algorytm dla drugiego podzadania wykonuje trochę więcej obliczeń niż to potrzeba dla wyznaczenia najkrótszego czasu wykonania przedsięwzięcia, ale dzięki temu odpowiedzi na pojedyncze pytanie w trzecim podzadaniu można udzielić w stałym czasie, tj. w czasie, który nie zależy od rozmiarów sieci.

Wykonalność przedsięwzięcia

Zgodnie z definicją podaną w treści zadania, przedsięwzięcie jest **wykonalne**, jeśli żaden podzbiór czynności nie tworzy cyklu w sensie relacji poprzedzania. Jest to uzasadnione tym, że gdyby istniał cykl czynności, to ponieważ czas wykonanie każdej czynności jest większy od zera, żadnej z czynności tego cyklu nie możnaby rozpocząć, czyli całe przedsięwzięcie nie byłoby wykonalne.

Metoda sprawdzania, czy przedsięwzięcie zdefiniowane za pomocą relacji poprzedzania nie zawiera cyklu opiera się na dwóch bardzo prostych własnościach relacji poprzedzania.

Własności

1. *Każde wykonalne przedsięwzięcie zawiera czynność, która nie jest poprzedzana przez żadną inną czynność – jest to czynność, od której można rozpocząć wykonywanie przedsięwzięcia.*

2. *Po usunięciu tej czynności z przedsięwzięcia wykonalnego pozostaje również przedsięwzięcie wykonalne.*

W terminach digrafów te własności oznaczają, że digraf acykliczny (czyli taki, w którym nie ma cykli) zawiera wierzchołek, którego stopień wejściowy jest równy 0 i po usunięciu tego wierzchołka z digrafu, pozostały digraf jest także acykliczny. Możemy więc kontynuować ten proces usuwania z digrafu wierzchołków, które w redukowanym digrafie stają się wierzchołkami stopnia zero, aż do wyczerpania całego zbioru wierzchołków. Jest to możliwe wtedy i tylko wtedy, gdy digraf jest acykliczny. Ustawmy wierzchołki w kolejności

$w_1, w_2, \ldots, w_N$, w jakiej zostały usunięte z digrafu. Zauważmy następującą ciekawą własność tego uporządkowania: ponieważ stopień wejściowy usuwanego w danej chwili wierzchołka jest równy 0, wszystkie wierzchołki, które poprzedzają go w digrafie (a więc wszystkie czynności, które poprzedzają w przedsięwzięciu czynność odpowiadającą usuwanemu wierzchołkowi) zostały usunięte z digrafu przed nim. Takie uporządkowanie wierzchołków acyklicznego digrafu nazywamy **uporządkowaniem topologicznym**.

Uzasadniliśmy powyżej prawdziwość następującego twierdzenia;

Twierdzenie. *Digraf jest acykliczny wtedy i tylko wtedy, gdy jego wierzchołki można uporządkować topologicznie, czyli ustawić w takim porządku* $w_1, \ldots, w_N$, *że*

$$\textit{jeśli } (w_i, w_j) \textit{ jest łukiem w tym digrafie, to } i < j.$$

Zatem aby sprawdzić, czy przedsięwzięcie jest wykonalne, należy zbadać czy reprezentujący go digraf jest acykliczny. Będziemy to sprawdzać generując jednocześnie uporządkowanie topologiczne jego wierzchołków, które wykorzystamy w algorytmie dla drugiego kroku rozwiązania zadania, czyli dla określenia najkrótszego czasu wykonania całego przedsięwzięcia, jeśli tylko jest ono wykonalne.

Algorytm sukcesywnego usuwania z digrafu wierzchołków o stopniu wejściowym zero jest zaprogramowany w funkcji logicznej `TopologiczneSortowanie`, która przyjmuje wartość `True`, gdy digraf jest acykliczny i `False` – w przeciwnym razie. W tym pierwszym przypadku dodatkowym wynikiem obliczenia wartości funkcji jest topologiczne uporządkowanie wierzchołków digrafu zapisane w tablicy `TopolSort`. W procedurze tej skorzystaliśmy z dość prostego spostrzeżenia, które ułatwia znajdowanie kolejnego wierzchołka nie poprzedzanego przez żaden z pozostałych jeszcze wierzchołków. Otóż na początku znajdujemy wszystkie wierzchołki, które nie mają poprzedników w digrafie i umieszczamy w tablicy `TopolSort`, a następnie dopisujemy do tej tablicy te wierzchołki, które tracą poprzedników w wyniku redukcji digrafu.

```
function TopologiczneSortowanie(N:NumWierzch;Dig:Digraf;
                                var TopolSort:Wektor):Boolean;
 var i,j,k,kk,l:Integer;
     Cykl       :Boolean;
     p          :PDigrafElem;
     a          :Wektor;
begin
 for i:=1 to N do
  a[i]:=Dig[i].WeStopien;
 k:=0;  l:=0;
 for i:=1 to N do
```

```
  if a[i]=0 then begin
   k:=k+1;
   TopolSort[k]:=i
  end;
 Cykl:=k>0;
 while (k<N) and Cykl do begin
  kk:=k;
  for j:=l+1 to kk do begin
   p:=Dig[TopolSort[j]].WyLista;
   while p<>nil do begin
    i:=p^.Wierzch;
    a[i]:=a[i]-1;
    if a[i]=0 then begin
     k:=k+1;
     TopolSort[k]:=i
    end;
    p:=p^.Nastepny
   end {while p<>nil}
  end; {for}
  l:=kk;
  Cykl:=k<>kk
 end; {while (k<N) ...}
 TopologiczneSortowanie:=Cykl
end; {TopolgiczneSortowanie}
```

Określmy złożoność tego kroku rozwiązania. Na początku przeglądamy wszystkie wierzchołki w poszukiwaniu tych, które nie mają poprzedników. Następnie wykonujemy N razy usunięcie wierzchołka z digrafu i ta operacja jest związana z redukcją stopnia wejściowego dla wszystkich bezpośrednich następników usuwanego wierzchołka. Zatem usunięcie wszystkich wierzchołków jest związane z wykonaniem liczby operacji równej sumarycznej liczbie wszystkich łuków wychodzących z wierzchołków digrafu, czyli liczby wszystkich łuków digrafu – oznaczmy tę liczbę przez M. W rezultacie liczba wykonanych operacji w tym kroku rozwiązania jest proporcjonalna do $N+M$.

Najkrótszy czas wykonania całego przedsięwzięcia

Gdyby wszystkie czynności przedsięwzięcia miały być wykonane jedna po drugiej, na przykład w porządku topologicznym, to czas takiego wykonania całego przedsięwzięcia byłby równy sumie czasów wykonania wszystkich czynności i ... nie byłoby żadnego zadania do rozwiązania. W wielu przypadkach nie jest to jednak najkrótszy czas wykonania przedsięwzięcia, gdyż na ogół wiele

czynności można wykonywać jednocześnie. Istotny wpływ na czas wykonania całego przdsięwzięcia mają te podzbiory czynności, które muszą być wykonane jedna po drugiej, a więc czynności tworzące drogę w digrafie odpowiadającym przedsięwzięciu, i oczywiście przedsięwzięcia nie można ukończyć wcześniej niż wynosi czas wykonania jakiegokolwiek ciągu następujących po sobie czynności. Zatem w sieci reprezentującej przedsięwzięcie należy znaleźć długość najdłuższej drogi w sensie sumarycznego czasu wykonania tworzących ją czynności.

Algorytm znajdowania długości najdłuższej drogi w sieci w istotny sposób korzysta z własności sieci, że jej digraf jest acykliczny. W sieci opartej na dowolnym digrafie najdłuższa droga może nie istnieć, a dokładniej – może mieć nieskończoną długość. W digrafie acyklicznym nie ma nieskończonych dróg, wszystkie drogi są skończone, zatem istnieje droga o największej długości i jest to najkrótszy możliwy czas wykonania całego przedsięwzięcia. Metoda jaką zastosujemy wyznacza długość najdłuższej drogi w sieci acyklicznej poprzez znalezienie długości najdłuższych dróg kończących się w kolejnych wierzchołkach, zgodnie z uporządkowaniem topologicznym. Dla ustalonego wierzchołka, długość najdłuższej drogi kończącej się w tym wierzchołku jest równa maksimum po wszystkich długościach dróg dochodzących do bezpośrednich poprzedników tego wierzchołka plus czas wykonania czynności w tym wierzchołku. Czas wykonania czynności na najdłuższej drodze kończącej się w danym wierzchołku c (a dokładniej czynnością c) jest oznaczony przez `MinCzasZak[c]`. Zauważmy, że ten czas pomniejszony o czas wykonania czynności c jest najwcześniejszym możliwym terminem rozpoczęcia wykonywania czynności c, gdyż aby zacząć jej wykonywanie musimy zaczekać aż zostaną wykonane wszystkie czynności ją poprzedzające w przedsięwzięciu. Wartością funkcji `CzasPrzedsiewziecia` jest najkrótszy czas potrzebny do wykonania wszystkich czynności przedsięwzięcia z zachowaniem relacji poprzedzania między czynnościami.

W funkcji `CzasPrzedsiewziecia` zawarliśmy ponadto liczenie dla każdej czynności c wielkości `MaxCzasZak[c]`, która jest najpóźniejszym czasem rozpoczęcia wykonywania wszystkich czynności przedsięwzięcia, następujących po czynności c. Zatem wielkość `MaxCzasZak[c]` pomniejszona o czas wykonania czynności c jest najpóźniejszym możliwym terminem rozpoczęcia wykonywania czynności c tak, aby całe przedsięwzięcie mogło być wykonane w możliwie najkrótszym czasie.

```
function CzasPrzedsiewziecia(N:NumWierzch;Dig:Digraf;
                             TopolSort:Wektor;
                    var MinCzasZak,MaxCzasZak:Wektor):Integer;
 var i,j,k,r,t,u:Integer;
     p          :PDigrafElem;
```

```
begin
 k:=0;
 repeat
  k:=k+1;
  MinCzasZak[TopolSort[k]]:=Dig[TopolSort[k]].Czas
 until (k=N) or (Dig[TopolSort[k+1]].WeStopien>0);
 for i:=k+1 to N do
  MinCzasZak[TopolSort[i]]:=0;
 for i:=k+1 to N do begin
  j:=TopolSort[i];
  p:=Dig[j].WeLista;
  r:=0;
  while p<>nil do begin
   t:=MinCzasZak[p^.Wierzch];
   if t>r then r:=t;
   p:=p^.Nastepny
  end;
  MinCzasZak[j]:=r+Dig[j].Czas
 end; {for i:=k+1}
 t:=MinCzasZak[1];
 for i:=2 to N do
  if t<MinczasZak[i] then
   t:=MinCzasZak[i];
 CzasPrzedsiewziecia:=t;
 for i:=N downto 1 do
  if Dig[i].WyStopien=0 then MaxCzasZak[i]:=t
  else MaxCzasZak[i]:=0;
 for i:=N downto 1 do begin
  j:=TopolSort[i];
  p:=Dig[j].WyLista;
  r:=t;
  while p<>nil do begin
   u:=MaxCzasZak[p^.Wierzch]-Dig[p^.Wierzch].Czas;
   if u<r then r:=u;
   p:=p^.Nastepny
  end;
  MaxCzasZak[j]:=r
 end {for i:=N}
end; {CzasPrzedsiwziecia}
```


Złożoność tego kroku rozwiązania jest również proporcjonalna do $N + M$, gdyż wszystkie wykonywane operacje można pogrupować względem wierzchołków, a w wierzchołkach – względem wchodzących i wychodzących łuków, których jest M.

Wydłużanie terminu zakończenia całego przedsięwzięcia

W tym kroku rozwiązania skorzystamy z wielkości w tablicy `MaxCzasZak`, policzonych w poprzednim kroku. Zauważmy, że jeśli `MinCzasZak[c]=MaxCzasZak[c]`, to chcąc ukończyć całe przdsięwzięcie w możliwie najkrótszym czasie musimy rozpocząć wykonywanie czynności `c` natychmiast po zakończeniu wykonywanie czynności ją bezpośrednio poprzedzających. Taką czynność nazywamy **krytyczną**, gdyż chcąc zakończyć całe przedsięwzięcie w możliwie najkrótszym czasie nie można ani opóźnić rozpoczęcia, ani przedłużyć jej wykonywania. Jeśli czynność nie jest krytyczna, to mamy pewną swobodę (luz) na jedno lub na drugie. Zatem dla czynności `c`, jej czas wykonywania można przedłużyć o co najwyżej

```
MaxCzasZak[c]-MinCzasZak[c]
```

nie ryzykując wydłużenia najkrótszego czasu wykonania całego przedsięwzięcia.

Zatem odpowiedź na jedno pytanie w trzecim kroku algorytmu, dzięki wielkościom wyznaczonym w drugim kroku, może być udzielona w stałym czasie, tzn. niezależnym od wielkości sieci.

Fragment programu, w którym wykorzystano opisane wyżej struktury danych i procedury ma postać:

```
 Read(N);
{Ustalenie wartosci poczatkowych w strukturach danych.}
 for i:=1 to N do
  with Dig[i] do begin
   WeLista:=nil;  WyLista:=nil;
   WeStopien:=0;  WyStopien:=0
  end;
{Czytanie nieco zmienionych danych.}
 for i:=1 to N do begin
  Read(c);
  CzytajOpisCzynnosci(c)
 end;
{Wlasciwe obliczenia.}
 if not TopologiczneSortowanie(N,Dig,TopolSort) then
  Write('Siec przedsiewziecia zawiera cykl')
```

```
  else begin
    t:=CzasPrzedsiewziecia(N,Dig,TopolSort,MinCzasZak,MaxCzasZak)
{Tutaj mozna umiescic sprawdzanie, czy wydluzenie czasu
wykonania czynnosci nie wydluzy czasu wykonania calego
przedsiewziecia - wystarczy w tym celu posluzyc sie
wartosciami z tablic MinCzasZak i MaxCzasZak.}
```

Inne rozwiązania tego zadania są krótko przedstawione poniżej w omówieniu rozwiązań uczniów. Większość innych algorytmów nie gwarantuje tak niskiej złożoności obliczeniowej, jak opisana przez nas metoda. Poza tym, są to zwykle metody bardziej złożone, zarówno w sensie użytych struktur danych, opisu algorytmów, jak i ich implementacji.

Omówienie rozwiązań podanych przez uczniów

Jak nadmieniliśmy wcześniej, zgodnie z przyjętym regulaminem przeprowadzania zawodów II i III stopnia, przez 30 minut od wręczenia uczniom treści zadania mogą oni kierować do dyżurujących członków Komitetu Głównego pytania na piśmie dotyczące niejasnych dla nich sformułowań w treści zadania.

Najwięcej wątpliwości mieli uczniowie w związku z treścią zadania PRZEDSIĘWZIĘCIE i ich pytania dotyczyły bardzo istotnych dla rozwiązania kwestii: czy czynności mogą być wykonywane jednocześnie (czyli równolegle) i jeśli tak, jaki wpływ na czas wykonania całego przedsięwzięcia mają czasy wykonywania czynności równoległych (oczywiście pytania tego drugiego rodzaju były formułowane tak, aby zgodnie z regulaminem móc uzyskać na nie odpowiedź TAK lub NIE).

Odpowiedź na te wątpliwości jest do pewnego stopnia zawarta w treści zadania. Zauważmy bowiem, jeśli przedsięwzięcie jest wykonalne, czyli nie zawiera cyklu czynności, to najkrótszy czas wykonania wszystkich czynności jedna po drugiej i tak aby żadne dwie nie były wykonywane jednocześnie jest równy sumie czasów wykonania wszystkich czynności, a zatem rozwiązanie byłoby bardzo proste.

Można przypuszczać, że początkowe trudności związane z właściwym zrozumieniem treści zadania mogły mieć głębsze podłoże – otóż człowiek myśli sekwencyjnie (chociaż procesy w mózgu zachodzą współbieżnie) i większość algorytmów programujemy tak, aby były wykonywane szeregowo – ze względu na komputery, jakimi się posługujemy. Rozwiązania tego zadania podane przez uczniów wskazują jednak, że ta bariera, jeśli istniała u niektórych, została szybko pokonana. Ze względu jednak na liczbę osób zadających takie samo

pytanie, treść zadania uzupełniono i podano do wiadomości wszystkim zawodnikom, że czynności przedsięwzięcia mogą być wykonywane równocześnie.

W większości rozwiązań uczniowskich przedsięwzięcie było reprezentowane podobnie jak w naszym rozwiązaniu powyżej, czyli w postaci tablicy rekordów, po jednym dla każdej czynności. Kilku uczniów użyło obiektów, ale w tym przypadku nie było to konieczne.

Metody zastosowane przez uczniów w rozwiązaniach do sprawdzenia wykonalności przedsięwzięcia i obliczenia najwcześniejszego czasu jego wykonania można podzielić na cztery grupy:

A. Pierwszą grupę tworzą metody polegające na obliczaniu najwcześniejszego czasu zakończenia całego przedsięwzięcia przez analizowanie a później usuwanie czynności, które mogą być wykonane ponieważ wykonane zostały już wszystkie czynności je poprzedzające w przedsięwzięciu. A więc są to metody podobne do zaproponowanej powyżej.

Niewielu jednak uczniów osobno sprawdzało najpierw wykonalność przedsięwzięcia, a później wykonywało odpowiednie obliczenia czasów. Najczęściej stosowano rozumowanie: można obliczyć czas wykonania wybranej czynności, jeśli zostały już wykonane czynności ją poprzedzające. Jeśli jedank nie było takiej czynności wśród czynności jeszcze nie wykonanych, to słusznie sygnalizowano istnienie cyklu. Tylko w kilku rozwiązaniach pojawiło się jawnie sortowanie topologiczne.

B. Do drugiej grupy rozwiązań można zaliczyć te, w których obliczenia zostały zaprojektowane z wykorzystaniem rekurencji, zarówno do sprawdzania wykonalności przedsięwzięcia, jak i do obliczenia najkrótszego czasu ukończenia całego przedsięwzięcia. Najczęściej procedura rekurencyjna liczyła najwcześniejszy czas wykonania wybranej czynności wywołując siebie samą do policzenia najwcześniejszych czasów wykonania czynności ją poprzedzających. Jeśli ten proces wywołań (poprawnie zaprogramowany) trafiał na czynność już odwiedzoną w tym wywołaniu, to słusznie wnioskowano, że istnieje cykl wśród czynności. W przeciwnym przypadku otrzymywano najwcześniejsze czasy ukończenia poszczególnych czynności wywołując tę procedurę rekurencyjną osobno dla każdej czynności. Można zauważyć, że rekurencja w tym przypadku może być dość kosztowna, gdyż wiele obliczeń jest niepotrzebnie powtarzanych. Na przykład, gdyby digraf przedsięwzięcia tworzył drogę o N czynnościach, to metoda z topologicznym sortowaniem wykonałaby N obliczeń czasów zakończenia wykonywania czynności, a metoda rekurencyjna – proporcjonalnie do N^2. Kilku uczniów zauważyło tę rozrzutność w działaniu metody rekurencyjnej i wprowadziło globalną tablicę czasów zakończenia wykonywania czynności, która była

tylko modyfikowana w kolejnych wykonaniach rekurencyjnych, zamiast wypełniania jej od nowa dla każdej czynności oddzielnie.

C. Trzecią grupę rozwiązań stanowiły programy, w których czynności były rozważane dosłownie na osi czasu. Była to niepotrzebna komplikacja rozwiązania, bo i tak nie można się obejść w nim bez wykrywania cyklu i prowadzenia obliczeń zgodnie z porządkiem wyznaczonym przez relację poprzedzania między czynnościami. Autorzy takiego podejścia najczęściej rozróżniali czynności: wykonane, aktualnie wykonywane i niezależne od nich. Wykrycie tych ostatnich jest problemem samym w sobie, dość trudnym algorytmicznie.

D. Podane zostały również rozwiązania, w których podstawowy problem został właściwie zidentyfikowany jako wyznaczanie najdłuższej drogi w sieci i zaproponowano dla jego rozwiązania odmianę metody, która dąży do spełnienia warunku trójkąta w sieci (zob. rozwiązanie zadania PRZEPUSTOWOŚĆ).

W dwóch ostatnich podejściach bardziej utrudnione było wykrywanie cykli wśród czynności.

We wszystkich podejściach, najkrótszy możliwy czas zakończenia całego przedsięwzięcia znajdowano jako najdłuższy czas zakończenia wykonania jego czynności.

Trzecia część rozwiązania zadania, polegająca na określenie czy zwiększenie czasu wykonania czynności będzie miało wpływ na przedłużenie terminu zakończenia całego przedsięwzięcia, w zdecydowanej większość rozwiązań uczniowskich zostało zrealizowane następująco:

1. Oblicz najwcześniejszy czas wykonania całego przedsięwzięcia dla oryginalnych danych o przedsięwzięciu.
2. Dla podanej czynnosci zwiększ jej czas wykonania o podaną wartość i dla tak zmienionych danych ponownie wykonaj krok 1.
3. Udziel odpowiedzi porównując najwcześniejsze czasy wykonania przedsięwzięć z kroku 1 i 2.

Jest to naturalne podejście, zwłaszcza w sytuacji ograniczonego czasu na znalezienie innego rozwiązania.

Tylko jedna praca zawierała rozwiązanie, w którym liczone były zapasy czasów dla poszczególnych czynności, tak jak zaproponowaliśmy w naszym rozwiązaniu.

Testy

Testy zastosowane do sprawdzenia rozwiązań uczniowskich miały na celu:

— zbadanie, czy program sprawdza wykonalność przedsięwzięcia, czyli czy bada istnienie cyklu w digrafie;

— sprawdzenie poprawności obliczeń czasu wykonania całego przedsięwzięcia nie zawierającego cyklu i wpływu na ten czas zmiany czasu wykonania jednej czynności;
— określenie efektywności zaproponowanych rozwiązań.

Digrafy przedsięwzięć niewykonalnych zawierały cykle w różnych konfiguracjach czynności. Niektóre przedsięwzięcia zawierały pojedyncze czynności lub grupy czynności niezależnych od innych.

Testy użyte do określenia efektywności rozwiązań były zbudowane na digrafach o dużej liczbie łuków tworzących gęstą konfigurację możliwych ciągów czynności. W szczególności jeden z testów był przedsięwzięciem złożonym z 1000 czynności i kilku tysięcy relacji poprzedzania między nimi.

7.2.3. Zadanie WYSPY NA TRÓJKĄTNEJ SIECI (Autor: Andrzej Walat)

W dalszej treści używamy skróconej nazwy tego zadania WYSPY.

Treść zadania WYSPY

Sieć trójkątów – patrz rysunek poniżej – jest zbudowana z równobocznych trójkątów o boku 1.

Ścieżką na sieci trójkątów nazywamy dowolny skończony ciąg trójkątów (o boku 1) sieci taki, że każde kolejne dwa trójkąty w tym ciągu mają wspólny bok.

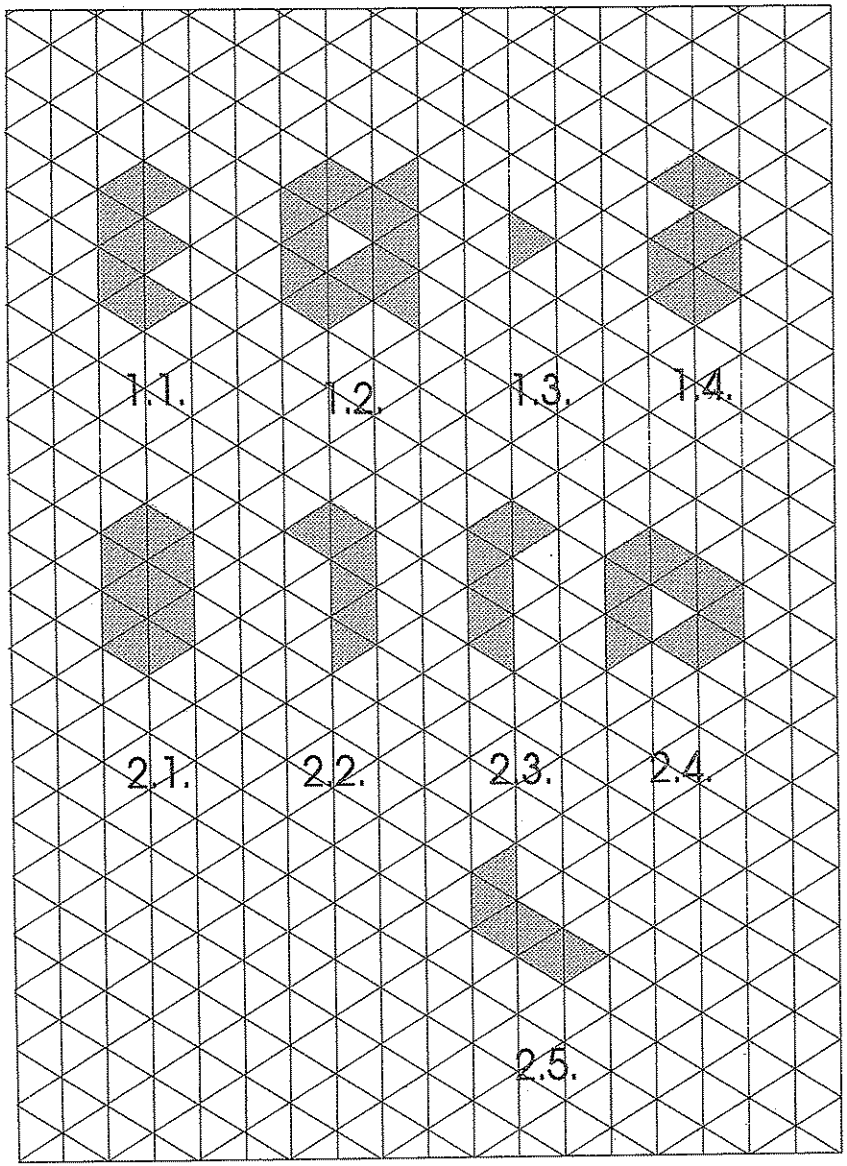
Figurę utworzoną przez punkty skończonej liczby trójkątów sieci nazywamy **wyspą**, jeśli dowolne dwa zawarte w niej trójkąty sieci można połączyć ścieżką, utworzoną z trójkątów zawartych w tej figurze.

Przykłady

Figury przedstawione na rysunkach 1.1, 1.2 i 1.3 są wyspami. Figura na rysunku 1.4 nie jest wyspą. Figury na rysunkach 2.2, 2.3 i 2.5 są przystające.

Zamierzamy dla każdego $n \leq 10$ opisać w systematyczny sposób wszystkie nieprzystające wyspy, jakie można utworzyć z n trójkątów o boku 1 i policzyć ile ich jest.

Brzeg każdej wyspy, zbudowanej z co najwyżej 10 trójkątów, jest łamaną zamkniętą, złożoną z jednostkowych odcinków siatki i można go obiec (obrysować bez odrywania ołówka od papieru), w ten sposób, że dokładnie jeden raz przebiegamy każdy odcinek brzegu i wracamy do punktu wyjścia. Może się zdarzyć, że trzeba będzie przy tym przejechać więcej niż jeden raz przez ten sam punkt – zob. np. rys. 2.4.



Figury utworzone z równobocznych trójkątów

W przypadku wysp zbudowanych z co najwyżej 10 trójkątów nie jest możliwa sytuacja taka, jak na rysunku 1.2, że brzeg tej figury składa się z dwóch rozłącznych części i nie można go obiec (obrysować bez odrywania ołówka od papieru).

Obiegając brzeg, po każdym jednostkowym odcinku wykonujemy skręt:

a – o 120 stopni w lewo, albo

b – o 60 stopni w lewo, albo

c – o 0 stopni (tzn. faktycznie nie wykonujemy skrętu), albo

d – o 60 stopni w prawo, albo

e – o 120 stopni w prawo.

Każdy obieg brzegu wyspy można opisać za pomocą słowa złożonego z liter ze zbioru {a, b, c, d, e} odnotowując za pomocą odpowiedniej litery skręt, jaki należy wykonać po każdym kolejnym, jednostkowym odcinku łamanej tworzącej brzeg. Opis obiegu brzegu wyspy ma tyle liter, z ilu jednostkowych odcinków składa się łamana tworząca ten brzeg. Odnotowujemy skręt także po ostatnim odcinku łamanej, chociaż nie jest to konieczne dla jednoznacznego określenia jej kształtu. Ta, w pewnym sensie nadmiarowa litera ułatwia przekształcenie danego opisu w opis innego obiegu tej samej figury, który zaczyna się w innym punkcie.

Przykłady

Słowa: cdddcddd, dcdddcdd oraz cbbbcbbb opisują różne obiegi figury na rysunku 2.1.

Słowa: cbeddcde, adcabcbb i abcbbadc opisują różne obiegi wyspy na rysunku 2.2.

Słowa: acdabbcb i cddebced opisują różne obiegi wyspy na rysunku 2.3.

Jeśli obiegając brzeg wyspy mamy stale jej wnętrze po prawej stronie, to mówimy, że jest to **obieg prawoskrętny**.

Dla każdej wyspy można wyznaczyć zbiór wszystkich wysp do niej przystających oraz ich obiegi prawoskrętne.

Kodem wyspy nazwiemy słowo, które:

S1. Jest opisem pewnego prawoskrętnego obiegu brzegu jakiejś wyspy do niej przystającej.

S2. Jest wcześniejsze w porządku alfabetycznym od wszystkich innych słów spełniających warunek S1.

Przykłady

Dla wysp przedstawionych na rysunkach 2.2 i 2.3, które są przystające, bierzemy pod uwagę wszystkie prawoskrętne obiegi obydwu:

beddcdec, eddcdecb, ddcdecbe, dcdecbed, cdecbedd, decbeddc, ecbeddcd, cbeddcde

oraz:

bcedcdde, cedcddeb, edcddebc, dcddebce, cddebced, ddebcedc, debcedcd, ebcedcdd

i jako kod każdej z nich wybieramy to słowo, które w porządku alfabetycznym należy umieścić na pierwszym miejscu: bcedcdde.

Kodem wyspy przedstawionej na rysunku 2.4 jest słowo: aadecddcddde.

Zadanie

Ułóż program, który kolejno dla każdej danej wejściowej z pliku tekstowego WYS.IN generuje zestaw wyników i zapisuje go w pliku tekstowym WYS.OUT.

Daną wejściową może być:

1. Słowo, które jest poprawnym kodem wyspy utworzonej przez co najwyżej 9 trójkątów, albo
2. Dodatnia liczba naturalna nie większa niż 10.

Plik WYS.IN jest zawsze niepusty i może zawierać wiele danych wejściowych. Kolejne dane są pisane w oddzielnych wierszach, bezpośrednio po ostatniej danej następuje koniec pliku.

Zakładamy, że plik WYS.IN jest bezbłędny i program nie musi sprawdzać jego poprawności.

Plik WYS.OUT powinien zawierać po jednym zestawie wyników dla każdej danej z pliku WYS.IN.

Mogą być dwa rodzaje zestawów wyników:

1. Jeśli daną wejściową jest kod k, to wynikiem będzie ciąg, którego pierwszym wyrazem jest liczba różnych kodów wysp, jakie można utworzyć z dowolnej z przystających wysp opisanych kodem k, przez dodanie jednego trójkąta o boku 1 (nie należącego do wyspy). Następnymi wyrazami są te kody, uporządkowane alfabetycznie. Kolejne wyrazy ciągu oddziela odstęp lub koniec wiersza.
2. Jeśli daną wejściową jest liczba n, to wynikiem ma być ciąg, którego pierwszym wyrazem jest liczba różnych kodów wysp, jakie można utworzyć z n trójkątów sieci o boku 1. Następnymi wyrazami są te kody, uporządkowane alfabetycznie. Kolejne wyrazy ciągu oddziela odstęp lub koniec wiersza.

Kolejne zestawy wyników są oddzielane pustym wierszem. Po ostatnim zestawie następuje koniec pliku.

Przykład

Dla pliku WYS.IN:


```
eee
adeccecced
5
dddddd
6 (koniec pliku)
```

plik WYS.OUT powinien zawierać:

```
1 dede
(pusty wiersz)
5
acedccecced addebcecced adebebecced adecbedcced cceccecce
(pusty wiersz)
4 aeddddе bdecdde bececde ccedcde
(pusty wiersz)
1 cddddce
(pusty wiersz)
12
adecddde aececdde aedcecde bcedcdde
bddebdde bdebecde bdeccece bdecdced bebedcde
becebece ccdeccde dddddd (koniec pliku)
```

Na początku programu podaj w komentarzu Twoje nazwisko, imię i numer stanowiska, na którym pracujesz.

Twój program powinien szukać pliku WYS.IN w katalogu bieżącym i tworzyć plik WYS.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę WYS.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku WYS.EXE. Oba te pliki powinny być zapisane na dysku stałym oraz na dyskietce.

Rozwiązanie zadania składa się z programu – tylko jednego – w postaci źródłowej i wykonywalnej, zapisanego zgodnie z podanymi wyżej warunkami na dysku stałym i dyskietce oraz opisu algorytmu rozwiązania wraz z uzasadnieniem jego poprawności.

Rozwiązanie zadania WYSPY

Procedura generowania dla danego kodu wyspy różnych kodów wszystkich wysp, jakie można z niej otrzymać przez dodanie jednego trójkąta

Aby poprawnie zdefiniować procedurę tworzenia uporządkowanej listy kodów wszystkich wysp, jakie można utworzyć z wyspy o danym kodzie przez

dodanie do niej jednego trójkąta, należy odpowiednio uwzględnić wiele różnych szczegółów.

Po pierwsze zauważmy, że wśród wysp złożonych z co najwyżej dziewięciu trójkątów jednostkowych znajduje się jedna wyjątkowa, otaczająca trójkątną zatokę. Jest to jedyna wyspa, której kod zaczyna się od dwóch liter a i potraktujemy ją jako wyjątek.

We wszystkich pozostałych przypadkach, kolejne wyspy pochodne trójkąta będziemy generowali przez dodanie jednego trójkąta do odpowiedniego boku danej wyspy.

W tym celu zdefiniujemy pomocnicze operacje na słowach używanych jako opisy (niekoniecznie kody) wysp oraz na listach słów.

Zakładamy, że stałe i niestandardowe typy danych występujące w omawianych poniżej procedurach zostały zdefiniowane w następujący sposób:

```
const
 N=13;
type
 Wyraz        =string[N];
 WskListy     =^ElementListy;
 ElementListy =record
                 Kod:Wyraz;
                 Wsk:WskListy
                end;
```

Przewijanie słowa (Przew) jest operacją (funkcją), której wynikiem jest dane słowo z przestawioną na koniec pierwszą literą.

```
function Przew(Obieg:Wyraz):Wyraz;
begin
 Przew:=Copy(Obieg,2,Length(Obieg)-1)+Obieg[1]
end; {Przew}
```

Doklejeniu jednego trójkąta do boku wyspy odpowiada operacja na słowach o nazwie Next. Jeśli Obieg jest słowem opisującym prawoskrętny obieg wyspy, to Next(Obieg) jest opisem zaczynającego się w tym samym punkcie obiegu wyspy pochodnej, utworzonej przez dołączenie trójkąta do drugiego boku (w kolejności danego obiegu) danej wyspy.

Na przykład Next('cedde')='beddde'.


```
function Next(Obieg:Wyraz):Wyraz;
begin
 if Obieg[2]='a' then Next:=
 Pred(Obieg[1])+Pred(Obieg[3])+Copy(Obieg,4,Length(Obieg)-3)
 else Next:=
 Pred(Obieg[1])+'e'+Pred(Obieg[2])+Copy(Obieg,3,Length(Obieg)-2)
end; {Next}
```

Obliczanie wartości funkcji Next według powyższej definicji polega na odpowiednim przekształceniu przedrostka danego słowa. Analizując skończoną liczbę przypadków łatwo sprawdzić, że daje ona zawsze poprawny wynik, o ile dany obieg nie zaczyna się od litery a. Będziemy jej używali tylko w takich przypadkach.

Wynikiem funkcji Next jest słowo, które na ogół jest poprawnym opisem jakiegoś prawoskrętnego obiegu wyspy, ale nie musi być kodem tej wyspy. Z tego powodu definiujemy funkcję PoprawnyKod, która na podstawie dowolnego obiegu wyspy znajduje jej kod.

```
function PoprawnyKod(Obieg:Wyraz):Wyraz;
 var MinWyraz:Wyraz;
     i       :Byte;
begin
 MinWyraz:=Obieg;
 for i:=2 to Length(Obieg) do begin
  Obieg:=Przew(Obieg);
  if Obieg<MinWyraz then MinWyraz:=Obieg
 end;
 Obieg:=Wspak(Obieg);
 for i:=1 to Length(Obieg) do begin
  Obieg:=Przew(Obieg);
  if Obieg<MinWyraz then MinWyraz:=Obieg
 end;
 PoprawnyKod:=MinWyraz
end; {PoprawnyKod}
```

Obliczanie wartości funkcji PoprawnyKod polega na przejrzeniu opisów wszystkich obiegów danej wyspy oraz jej symetrycznego odbicia i wybraniu słowa najwcześniejszego w porządku alfabetycznym.

Obieg wyspy symetrycznej znajdujemy biorąc obieg danej wyspy wspak. Funkcję Wspak można zdefiniować w następujący sposób:

```
function Wspak(Obieg:Wyraz):Wyraz;
begin
 if Length(Obieg)=1 then Wspak:=Obieg
 else Wspak:=Wspak(Copy(Obieg,2,Length(Obieg)-1))+Obieg[1]
end; {Wspak}
```

Na koniec zdefiniujemy niezbędne procedury działań na listach słów.

Zadaniem procedury Dopisz jest dopisanie do uporządkowanej alfabetycznie, niepustej listy różnych słów, wskazywanej przez wskaźnik Pochodne, nowego słowa – wartości parametru Element, w taki sposób, aby zachowany był porządek alfabetyczny słów tej listy. Nowy element jest dopisywany tylko wtedy, gdy nie występuje w liście – w takim przypadku jest również zwiększany Licznik o 1.

```
procedure Dopisz(Element:Wyraz;var Pochodne:WskListy;
                 var Licznik:Byte);
 var Poprz,Biez,Nowy:WskListy;
begin
 Biez:=Pochodne;
 while (Biez<>nil) and (Element>Biez^.Kod) do begin
  Poprz:=Biez;  Biez:=Biez^.Wsk
 end;
 if Element<>Biez^.Kod then begin
  New(Nowy);
  Nowy^.Kod:=Element;
  Nowy^.Wsk:=Biez;
  Poprz^.Wsk:=Nowy;
  Licznik:=Licznik+1
 end
end; {Dopisz}
```

Procedura Generuj dla danego kodu wyspy – wartości parametru Obieg – tworzy uporządkowaną alfabetycznie listę kodów wysp pochodnych, dostępną następnie poprzez wskaźnik Pochodne, a także wyznacza ich liczbę i przypisuje ją zmiennej LWysp.

Użyty algorytm, kolejno dla wszyskich prawoskrętnych obiegów danej wyspy, jakie można otrzymać przez cykliczne przewijanie danego obiegu, znajduje odpowiednią wyspę pochodną doklejając trójkąt (zawsze do drugiego boku bieżącego obiegu) i jeśli kod tej wyspy jeszcze nie wystąpił, to dopisuje go do listy kodów wysp pochodnych, zwiększając odpowiednio ich licznik. Jeśli bieżący

obieg zaczyna się od litery a, to pomijamy wyznaczanie wyspy pochodnej, ponieważ jeśli pomiędzy dwoma kolejnymi bokami prawoskrętnego obiegu jest wykonany skręt w lewo o 120 stopni, to dołączenie trójkąta do każdego z tych dwóch boków daje taką samą wyspę i wystarczy wykonać tę operację tylko raz.

Jeżeli dany obieg zaczyna się od przedrostka aa, to stosujemy procedurę specjalną, odpowiednią dla tego przypadku.

```
procedure Generuj(Obieg:Wyraz;var Pochodne:WskListy;
                  var LWysp:Byte);
 var i,n:Byte;
begin
 if Copy(Obieg,1,2)='aa' then
  SpecjalnyPrzypadek(Pochodne,LWysp)
 else begin
  n:=Length(Obieg)-1;
  if Obieg[1]='a' then Obieg:=Przew(Obieg);
  New(Pochodne);
  Pochodne^.Kod:=PoprawnyKod(Next(Obieg));
  Pochodne^.Wsk:=nil;
  Obieg:=Przew(Obieg);
  for i:=1 to n do begin
   if Obieg[1]<>'a' then
    Dopisz(PoprawnyKod(Next(Obieg)),Pochodne,LWysp);
   Obieg:=Przew(Obieg)
  end
 end {else}
end; {Generuj}
```

Należy jeszcze zdefiniować procedurę SpecjalnyPrzypadek:

```
procedure SpecjalnyPrzypadek(var Pochodne:WskListy;
                                 LWysp:Byte);
 const A:array [1..5] of Wyraz=('aadecddcddde',
                                'aaececdcddde',
                                'aaedceccddde',
                                'aaeddcebddde',
                                'cdddcddd');
 var Koniec,Nowy:WskListy;
     i          :Byte;
begin
```

```
 New(Koniec);
 Koniec^.Kod:=A[1];  Koniec^.Wsk:=nil;
 Pochodne:=Koniec;
 for i:=2 to 5 do begin
  New(Nowy);
  New^.Kod:=A[i];  New^.Wsk:=nil;
  Koniec^.Wsk:=Nowy;
  Koniec:=Nowy
 end;
 LWysp:=5
end; {SpecjalnyPrzypadek}
```

Algorytm rozwiązania zadania, dla dowolnych dopuszczalnych danych słów lub liczb

W pliku WYS.IN są dwa rodzaje danych: kody wysp oraz liczby. Algorytm rozwiązania zadania w pierwszym przypadku, gdy dany jest kod wyspy, został opisany powyżej w postaci procedury `Generuj`.

W drugim przypadku, dla małych $n \in [1, 3]$ wynikiem jest odpowiednia lista jednoelementowa, np. dla $n = 3$ mamy cedde.

Dla $n = 4$ odpowiednia lista kodów wszystkich wysp o polu 4 trójkątów jednostkowych składa się z trzech słów: beddde, cdecde, cecece.

Dla $n \in [5, 10]$ odpowiednią listę różnych kodów wszystkich wysp, jakie można ułożyć z n trójkątów, można wygenerować z odpowiedniej listy kodów wysp utworzonych z $n - 1$ trójkątów. Należy w tym celu, dla każdego słowa z listy kodów wysp o polu $n - 1$, wyznaczać kody wszystkich wysp pochodnych i wpisywać je kolejno do pustej na początku listy w taki sposób, aby zachowany był porządek alfabetyczny i żadne słowo nie pojawiało się więcej niż jeden raz.

Jeśli w pliku WYS.IN jest wiele danych liczbowych, to program tworzący listę kodów wysp o ustalonym polu, będzie wielokrotnie wykonywał te same obliczenia (operacje). Dlatego warto napisać na początku pomocniczy program, generujący dla $n = 5, \ldots, 10$ listy kodów wysp o polu n i zapisujący je w odpowiednim pliku tekstowym. Listy te są dość krótkie, np. dla $n = 9$ odpowiednia lista składa się ze 166 słów, a dla $n = 10$ – ze 144 słów. Następnie, utworzone w ten sposób pliki można wpisać do programu rozwiązującego zadanie konkursowe, np. w postaci stałych tablic. Taki program, po wczytaniu liczby z pliku danych WYS.IN, będzie przepisywał do pliku WYS.OUT gotową tablicę i dzięki temu może rozwiązać zadanie szybciej niż zrobią to inne programy.

Omówienie rozwiązań podanych przez uczniów

Największą trudność sprawiało uczniom bezbłędne opisanie procedury generującej listy kodów różnych wysp, jakie można utworzyć przez dodanie jednego trójkąta. Błędy w tej procedurze ujawniały się nie tylko wtedy, kiedy dany był kod wyspy, ale również w przypadku danych liczbowych.

Najczęstsze błędy polegały na tym, że była nieodpowiednia liczba słów będących różnymi kodami wysp o ustalonym polu i lista zawierała powtórzenia lub braki. Ponadto, wyspy były reprezentowane na liście przez słowa nie będące kodami. Pojawiały się również różne dziwne słowa, zawierające znaki spoza dopuszczalnego alfabetu, zapewne na skutek nieuwzględnienia różnych szczególnych przypadków. Typowy błąd, w rozwiązaniach bliskich ideału, polegał na pojawieniu się na liście 448 różnych kodów wysp o polu 10 jakiegoś jednego „dziwoląga" pochodnego od aaedddeddde.

Testy

W siedmiu testach plik danych składał się z jednego wiersza zawierającego kod wyspy:

eee

dddddd

aaedddcddde

aebecebdde

bbedcebdde

ccccedcccde

aeaeddcdde

Celem tych testów było sprawdzenie, czy program rozwiązujący zadanie prawidłowo znajduje wyspy, jakie można otrzymać z wysp o danych kodach, w wybranych szczególnych przypadkach.

W pięciu testach jedyny wiersz pliku danych zawierał liczbę: 1, 3, 7, 9 i 10.

Jeden test zawierał pięć liczb w pięciu wierszach:

10

3

10

3

10

Celem tego testu było sprawdzenie, czy program znajduje rozwiązanie dla powtarzalnych danych tylko raz, czy liczy je za każdym razem od nowa.

7.3. Zawody III stopnia

7.3.1. Zadanie WAHANIA AKCJI NA GIEŁDZIE
(Autor: Piotr Chrząstowski-Wachtel)

W dalszej treści używamy skróconej nazwy tego zadania GIEŁDA.

Treść zadania GIEŁDA

Badając historię giełdy chcemy obliczyć maksymalny zysk (maksymalną różnicę pomiędzy ceną sprzedaży i ceną zakupu) jaki można było osiągnąć kupując, a następnie sprzedając pojedynczą akcję.

Sesje giełdy odbywają się codziennie. Liczba spółek jest zmienna, bo spółki mogą bankrutować; mogą się też pojawić nowe spółki. Każdego dnia dla każdej spółki ogłasza się cenę jej akcji w tym dniu lub informację o bankructwie. Spółka, która zbankrutowała, nie będzie już dalej notowana.

Giełda ma wieloletnią historię, ale liczba spółek odnotowanych równocześnie, w jednym dniu, nigdy nie spadła do 0 ani nie była większa niż 10000, chociaż łącznie w całej historii giełdy mogło ich być znacznie więcej.

Historia wahań cen akcji na giełdzie jest zapisana w pliku tekstowym WAH.IN. Każdy wiersz jest niepusty i opisuje wyniki jednej sesji; są w nim zapisane wahania (zmiany ceny akcji w stosunku do poprzedniego dnia) lub informacje o bankructwach dla każdej spółki obecnej na giełdzie. W pierwszym wierszu są to różnice pomiędzy cenami akcji spółek w drugim i w pierwszym dniu istnienia giełdy. W drugim wierszu różnice cen między trzecim i drugim dniem, itd.

Wahania cen akcji spółek giełdowych są odnotowywane zawsze w tej samej kolejności. Nazwy spółek pomijamy. Spółkę identyfikuje jej pozycja w wierszu określającym notowania danego dnia. Spółka, która wystąpi jako pierwsza w pierwszym wierszu, w kolejnych wierszach może się pojawić jedynie na pierwszym miejscu. Spółka druga w pierwszym wierszu będzie zawsze druga aż do czasu, gdy sama wypadnie z gry, bądź pierwsza spółka wypadnie z gry ustępując jej pierwszą pozycję itd.

Gdy pojawia się nowa spółka, to pierwsze wahanie ceny jej akcji odnotowuje się w pierwszym dniu po jej wejściu na giełdę. Nowe spółki dopisuje się zawsze na końcu odpowiedniego wiersza.

Jeżeli wiersz pliku nieoczekiwanie się wyczerpie nie dostarczając informacji o wszystkich spółkach, których występowania spodziewamy się na podstawie notowań poprzedniego dnia, to sytuację taką klasyfikujemy jako błąd danych. Identycznie klasyfikujemy sytuację, kiedy pierwszym notowaniem spółki jest informacja o jej bankructwie (liczba –99).

Wahnięcia akcji są odnotowywane jako liczby całkowite z przedziału [–99, 100]. Ujemna liczba, różna od –99 oznacza, że wartość akcji spadła, zero – że się nie zmieniła, zaś dodatnia – że odpowiednio wzrosła. Liczba –99 oznacza, że spółka zbankrutowała i nie będzie już dalej notowana.

Zadanie

Ułóż program, który dla zestawu danych o giełdzie z pliku WAH.IN zapisuje w pliku tekstowym WAH.OUT:

- słowo NONSENS, jeśli dane są niepoprawne,
- liczbę zero, gdy dane są poprawne, ale wszystkie akcje przynoszą straty, albo
- maksymalny zysk na pojedynczej akcji.

Przykład

Dla pliku WAH.IN

```
  1  –4   2   4
–99   2  –5   1
 –1   3  –2   5
  6  –2   1   1
 –1  –1   3   1 (koniec pliku)
```

Twój program powinien zapisać w pliku WAH.OUT liczbę 7.

Na początku programu podaj w komentarzu Twoje nazwisko, imię i numer stanowiska, na którym pracujesz.

Twój program powinien szukać pliku WAH.IN w katalogu bieżącym i tworzyć plik WAH.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę WAH.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Tena sam program w postaci wykonywalnej powinien być zapisany w pliku WAH.EXE. Oba te pliki powinny być zapisane na dysku stałym oraz na dyskietce.

Rozwiązanie zadania składa się z programu – tylko jednego – w postaci źródłowej i wykonywalnej, zapisanego zgodnie z podanymi wyżej warunkami na dysku stałym i dyskietce oraz opisu algorytmu wraz z uzasadnieniem jego poprawności.

Rozwiązanie zadania GIEŁDA

Gdyby notowane były akcje tylko jednej spółki, to rozwiązanie zadania sprowadzałoby się do znalezienia spójnego podciągu danych o maksymalnej sumie. Ze względu jednak na to, że spółek może być wiele, należy równolegle tworzyć rozwiązanie dla wszystkich z nich. Metoda przetwarzania w rodzaju: „prze-

analizuj każdą spółkę oddzielnie i wybierz tę, która osiągnęła największy zysk" prowadzi do bardzo nieefektywnego algorytmu, wymagającego wielokrotnego przetwarzania pliku wejściowego, w którym dane o spółkach są zapisane dzień po dniu, a nie spółka po spółce.

Rozwiązanie tego zadania nie mające powyższej wady może wzorcowo pokazać siłę zastosowania listowych struktur danych. W naszym przypadku będziemy tworzyli listę aktywnych na giełdzie spółek. Na początku jest to lista pusta, a następnie wczytujemy dane z kolejnych dni i aktualizujemy listę, w razie potrzeby wyrzucając z niej zbankrutowane spółki lub dodając na końcu listy spółki nowo wchodzące na giełdę. Zauważmy, że koszt wykonania każdej z tych operacji jest stały (tj. nie zależy od długości listy), gdyż wskaźnik jest zawsze umieszczony na właściwym miejscu – nie musimy więc przeszukiwać listy aby znaleźć element do usunięcia, bądź miejsce do wstawienia nowego elementu.

Zastanówmy się, co chcielibyśmy przechowywać w rekordzie reprezentującym spółkę? Narzuca się pomysł, aby znajdowała się tam historia wahnięć akcji spółki (np. w postaci listy), ale krótkie przyjrzenie się możliwym rozmiarom danych wejściowych powinno skłonić nas do szukania innego rozwiązania.

Zauważmy, że w istocie interesuje nas różnica między możliwie małym kursem akcji, a możliwie dużym po nim następującym (żaden z nich nie musi być maksymalnym lub minimalnym kurem w całej historii). Aby obliczyć bieżący kurs akcji brakuje nam jednej danej: kursu z pierwszego dnia, jednakże w rzeczywistości możemy się bez niego obejść, gdyż interesują nas wielkości względne, a te są niezależne od kursu dnia pierwszego. Możemy więc przyjąć, że pierwszego dnia wszystkie akcje kosztowały 0 – to będzie nasz stały punkt odniesienia.

Dla każdej spółki będziemy zatem potrzebowali jedynie dwóch wartości: minimalnej ceny jej akcji w całej dotychczasowej historii oraz bieżącej ceny akcji. Napotkawszy nową daną dotyczącą wahnięcia akcji tej spółki użyjemy jej do uaktualnienia wartości tych zmiennych.

Idea rozwiązania polega na ograniczeniu rozważań do jednego dnia. W globalnej zmiennej *Max* przechowujemy maksymalny zysk, jaki w historii giełdy można było osiągnąć (na początku $Max = -1$) i aktualizujemy listę spółek aktywnych danego dnia. W każdym rekordzie odpowiadającym danej spółce przechowujemy dwie wartości: minimalną cenę akcji tej spółki w całej historii (*MinCena*) oraz bieżącą wartość akcji (*Cena*). Dane dotyczące wahnięć kolejnego dnia wczytujemy po kolei. Dodajemy wahnięcie do pola *Cena*, przechowując w nim bieżącą cenę akcji. Różnicę między wartością tego pola a wartością *MinCena* dla danej spółki porównujemy z wartością zmiennej *Max* i uaktualniamy

ją w razie potrzeby. Jeżeli wartość *Cena* spada poniżej wartości *MinCena*, to aktualizujemy wartość tej ostatniej zmiennej.

Jeżeli wahnięcie wynosi –99, to usuwamy spółkę z listy aktywnych spółek, a jeżeli napotkamy na element w danych nie należący do bieżącej listy, to wydłużamy ją o nowy element przypisując polu *Cena* pobraną wartość z danych wejściowych, zaś polu *MinCena* – wartość 0.

Obliczenia rozpoczynamy z pustą listą spółek, a kończymy po wyczerpaniu się danych wejściowych w pliku WAH.DAT. Złożoność czasowa algorytmu jest liniowa, a pamięciowa – wynosi $3k+1$, gdzie $k \leq 10000$ oznacza maksymalną liczbę spółek odnotowanych jednego dnia ($2k$ – na dwa pola rekordu, k – na organizację listy i 1 na zmienną *Max*).

Ze względu na bardzo prostą postać, nie zamieszczamy żadnego fragmentu programu rozwiązującego to zadanie – Czytelnik zapewne potrafi samodzielnie zdefiniować typ wskaźnikowy, zadeklarować listę i opisać podstawowe operacje na niej, takie jak: zmiana wartości w polach jej elementów, dołączanie nowego elementu (na końcu) i usuwanie wskazywanego elementu (z dowolnego miejsca listy).

Omówienie rozwiązań podanych przez uczniów

Uczniowie potraktowali to zadanie na ogół dość luźno i w zasadzie nie wygenerowali żadnego optymalnego rozwiązania. Do najczęstszych usterek należały:

1. Próba przechowywania w pamięci wszystkich informacji zawartych w danych wejściowych.
2. Próba przetwarzania danych wejściowych spółka-po-spółce a nie dzień-po-dniu. Złożoność tego podejścia wynosi $O(mn^2)$, gdzie m jest liczbą dni, a n liczbą wszystkich spółek.
3. Użycie tablicy zamiast listy – wobec braku ograniczenia na liczbę spółek trudno jest przewidzieć rozmiar tablicy.
4. Użycie więcej niż $3k+1$ komórek pamięci (np. niepotrzebne zapamiętywanie maksymalnych zysków dla każdej spółki osobno).

Testy

Testowe dane wejściowe, dla których uruchamiano programy uczniów były następujących rodzajów:

Testy prawidłowe 1–4:

— 3-dniowy zapis dla 4 spółek;
— 5-dniowy zapis, początkowo dla 4 spółek; 1 spółka znika;
— 3-dniowy zapis, początkowo dla 4 spółek; 1 spółka powstaje;

— 3-dniowy zapis, początkowo dla 4 spółek; 1 spółka znika i 2 powstają drugiego dnia.

Testy z danymi nieprawidłowymi 5–8:

— 3-dniowy zapis, początkowo dla 4 spółek; 4 spółki znikają drugiego dnia i trzeciego dnia 4 nowe powstają (drugiego dnia giełda jest pusta);
— w pierwszym wierszu są 4 spółki, a w drugim wierszu na pozycji 5 pojawia się –99;
— w pierwszym wierszu pojawia się –99;
— w drugim wierszu jest o 1 element mniej niż w pierwszym;

Testy 9–11: Duże zestawy danych:

— 30000 spółek przez 6 dni – losowe dane ze zbioru {–1,0,1};
— 6 spółek przez 30000 dni – losowe dane ze zbioru {–1,0,1};
300 spółek przez 600 dni – losowe dane ze zbioru {–1,0,1}.

7.3.2. Zadanie ANAGRAMY (Autor: Wojciech Complak)

Treść zadania ANAGRAMY

Anagramy danego wyrazu powstają przez przestawianie liter. Na przykład: mając słowo „tuba", możemy przestawić litery „a" i „u" otrzymując wyraz „tabu", a zamieniając litery „b" i „t" otrzymujemy słowo „buta". Jeżeli litery powtarzają się w wyrazie, to ich zamiana nie daje nowego anagramu.

Każdy słownik, to znaczy skończony i niepusty ciąg wyrazów, można podzielić na klasy anagramów, zaliczając dwa wyrazy do tej samej klasy tylko wtedy, gdy są nawzajem swoimi anagramami.

Zadanie

Ułóż program, który dla danego słownika, którego wszystkie wyrazy są zapisane w pliku tekstowym ANA.IN, znajduje jego podział na klasy anagramów i zapisuje wyniki w pliku tekstowym ANA.OUT.

Każdy wyraz słownika jest zapisany w osobnym wierszu pliku ANA.IN i może być otoczony dowolną liczbą spacji lub tabulatorów. Wyrazy słownika składają się wyłącznie z małych liter alfabetu angielskiego od „a" do „z" i nie zawierają znaków narodowych.

Bezpośrednio po ostatnim wyrazie słownika następuje koniec pliku.

W słowniku może znajdować się do trzech tysięcy wyrazów, zapisanych w dowolnym porządku – niektóre mogą się powtarzać. Każdy wyraz może mieć długość do 30 znaków. W skrajnych przypadkach wszystkie wyrazy mogą należeć do jednej klasy albo każdy wyraz może tworzyć osobną klasę anagramów.

Zakładamy, że dane w pliku ANA.IN są zapisane bezbłędnie zgodnie z podanymi wyżej zasadami i Twój program nie musi tego sprawdzać.

Wyniki powinny być zapisane w pliku ANA.OUT zgodnie z następującymi zasadami:

- wszystkie wyrazy tworzące jedną klasę anagramów danego słownika powinny być zapisane w jednym wierszu, w porządku alfabetycznym, bez powtórzeń, oddzielane pojedynczym odstępem,
- kolejne klasy anagramów słownika powinny być zapisane w kolejnych wierszach pliku w ten sposób, aby ich pierwsze wyrazy tworzyły ciąg uporządkowany alfabetycznie,
- bezpośrednio po ostatnim wyrazie ostatniej klasy anagramów powinien być koniec pliku.

Przykład

Dla pliku ANA.IN:

liszka
tuba
tuba
klisza
kretes
anakonda
sekret
szalik
buta
tabu *(koniec pliku)*

Twój program powinien utworzyć następujący plik ANA.OUT :

anakonda
buta tabu tuba
klisza liszka szalik
kretes sekret *(koniec pliku)*

Na początku programu podaj w komentarzu Twoje nazwisko, imię i numer stanowiska, na którym pracujesz.

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

programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku ANA.EXE. Oba te pliki powinny być zapisane na dysku stałym oraz na dyskietce.

Rozwiązanie zadania składa się z programu – tylko jednego – w postaci źródłowej i wykonywalnej, zapisanego zgodnie z podanymi wyżej warunkami na dysku stałym i dyskietce oraz opisu algorytmu wraz z uzasadnieniem jego poprawności.

Rozwiązanie zadania ANAGRAMY

W poprawnym rozwiązaniu zadania kluczową rolę odgrywa algorytm sprawdzania przynależności wyrazu do określonej klasy anagramów. Najprostsza i najszybsza metoda klasyfikowania polega na obliczaniu sygnatur dla poszczególnych wyrazów i przyporządkowywaniu ich do klas według wyznaczonych sygnatur. Metody polegające na bezpośrednim porównywaniu liter w wyrazach, czy też na analizie wszystkich permutacji liter, nawet pomijając trudność implementacji, są skazane na niepowodzenie ze względu na ich złożoność czasową. Najczęściej, jako sygnaturę wyrazu (i klasy, do której należy) wykorzystuje się ciąg znaków tworzących ten wyraz, w którym poszczególne litery są posortowane, zazwyczaj alfabetycznie, w porządku rosnącym. W ten sposób otrzymuje się pierwszy – w sensie porządku leksykograficznego – element zbioru anagramów danego wyrazu.

Drugim ważnym czynnikiem decydującym o poprawności i efektywności rozwiązania jest sposób przechowywania danych w pamięci operacyjnej. Klasycznym rozwiązaniem jest lista list. Na podstawie słownika wejściowego może być utworzona lista podstawowa służąca do przechowywania informacji o klasach anagramów, w której z każdym jej elementem jest związana lista wyrazów należących do danej klasy. W przykładowym rozwiązaniu zastosowaliśmy jednak tablice dynamiczne, które upraszczają algorytmy wyszukiwania i wstawiania, umożliwiają wykorzystanie bibliotecznej procedury sortowania oraz pozwalają zredukować wielkość pamięci przeznaczonej na listy.

Stosunkowo łatwym zadaniem jest filtrowanie danych wejściowych. Dla każdego wiersza należy odrzucić wszystkie znaki spacji i tabulacji poziomej aż do napotkania małej litery alfabetu angielskiego. Wszystkie kolejne litery, aż po znak końca wiersza, końca pliku, spacji lub tabulacji tworzą wyraz (o długości nie większej niż 30 znaków) należący do słownika.

Przedstawiony poniżej program jest przykładowym rozwiązaniem zadania. Program, napisany w języku C, rozpoczyna się dyrektywami preprocesora włączającymi standardowe pliki nagłówkowe. Maksymalna długość wyrazu wejściowego jest zdefiniowana jako stała symboliczna MAX_LEN. Duże znaczenie ma

stała MEM_STEP. Określa ona wielkość przyrostu, z jakim rozrastają się tablice dynamiczne i jest dobierana na zasadzie kompromisu między optymalnym wykorzystaniem pamięci operacyjnej (MEM_STEP=1) a szybkością działania programu, na którą duży wpływ mają wywołania funkcji przydzielania i zwalniania pamięci dynamicznej.

Każda klasa anagramów jest reprezentowana jako struktura typu AClass, której elementami składowymi są:

— wskaźnik do łańcucha znaków sygnatury (`char *Signature`),
— rozmiar (dynamiczny) tablicy (`int NoOfWSlots`),
— liczba wyrazów w tablicy (`int NoOfWords`),
— wskaźnik do tablicy wskaźników do wyrazów (`char **List`).

Funkcja GetWord jest implementacją filtru wejściowego. Przekazuje ona kolejno rozpoznawane wyrazy w zmiennej globalnej Buf.

Kolejne wyrazy i klasy są pamiętane w tablicy wskazywanej przez zmienną globalną L – jest to realizowane w procedurze InsertWord. Na początku jest obliczana sygnatura klasy anagramów, do której należy dany wyraz i następnie są przeszukiwane dotychczas utworzone klasy. Jeśli brak jest klasy o danej sygnaturze, to jest tworzona nowa klasa anagramów z tą sygnaturą. Jeśli sygnatura zostanie odnaleziona, to w tablicy wyrazów z nią skojarzonej jest poszukiwany ostatnio wczytany wyraz. Jeśli go brak, to zostaje tam wstawiony.

Zakończenie przetwarzania pliku wejściowego następuje po wczytaniu znaku końca pliku EOF – filtr danych wejściowych przekazuje wówczas puste słowo. Następnie, za pomocą standardowej procedury bibliotecznej szybkiego sortowania są porządkowane alfabetycznie tablice słów należących do poszczególnych klas, a na końcu tablica klas. Tak przetworzone i przygotowane dane są wyprowadzane do pliku wyjściowego.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_LEN 30
#define MEM_STEP 16

struct AClass
{
 char *Signature;
 int NoOfWSlots, NoOfWords;
 char **List;
}*L=NULL;
```

```
char Buf[MAX_LEN+1];
FILE *In, *Out;
int NoOfCSlots=0, NoOfClasses=0;

void GetWord(void)
{
 int i=0, c;
 for(;;)
  switch(c = getc(In))
  {
   case(EOF)  : Buf[i]='\0';
                return;
   case('\t') :
   case('\n') :
   case(' ')  : if(!i)continue;
                Buf[i]='\0';
                return;
   default    : Buf[i++]=(char)c;
  }
}
int ccSortFun(const void *a, const void *b)
{
 char ca = *(const char *)a, cb = *(const char *)b;
 return(ca-cb);
}
void InsertWord(void)
{
 int Len = strlen(Buf);
 char *Sig = malloc(Len+1);
 int i, j;
 struct AClass *P;
 qsort(strcpy(Sig,Buf),Len,sizeof(char),ccSortFun);
 for(i=0,P=L;
     (i<NoOfCSlots)&&(P->Signature!=NULL);
     i++,P++)
  if(!strcmp(P->Signature,Sig))
  {
   free(Sig);
   for(j=0;(j<P->NoOfWSlots)&&(*(P->List+j)!=NULL);j++)
```



```
    if(!strcmp(Buf,*(P->List+j)))return;
   if(j==P->NoOfWSlots)
   {
    P->List=realloc(P->List,(P->NoOfWSlots+=MEM_STEP)
                    *sizeof(char *));
    for(i=j;i<P->NoOfWSlots;i++)*(P->List+i)=NULL;
   }
   *(P->List+j)=strcpy(malloc(Len+1),Buf);
   P->NoOfWords++;
   return;
  }
 if(i==NoOfCSlots)
 {
  P=L=realloc(L,(NoOfCSlots+=MEM_STEP)*sizeof(struct AClass));
  for(j=i;j<NoOfCSlots;j++)
  {
   (P+j)->List=NULL;
   (P+j)->Signature=NULL;
   (P+j)->NoOfWSlots=0;
   (P+j)->NoOfWords=1;
  }
 }
 (L+i)->Signature=Sig;
 NoOfClasses++;
 (L+i)->List=malloc(((L+i)->NoOfWSlots=MEM_STEP)
                    *sizeof(char *));
 for(j=0;j<MEM_STEP;j++)*((L+i)->List+j)=NULL;
 *((L+i)->List)=strcpy(malloc(Len+1),Buf);
}
int ssSort_Fun(const void *a, const void *b)
{
 return(strcmp(*(char **)a,*(char **)b) );
}
int csSort_Fun(const void *a, const void *b)
{
 return(strcmp(((struct AClass *)a)->Signature,
               ((struct AClass *)b)->Signature));
}
void main(void)
```

```
{
 int i, j;
 struct AClass *P;
 In = fopen("ANA.IN", „rt");
 Out = fopen("ANA.OUT","wt");
 for(GetWord();strcmp(Buf,"");InsertWord(),GetWord());
 fclose(In);
 for(i=0,P=L;(i<NoOfCSlots)&&(P->Signature!=NULL);P++,i++)
  qsort((void *)P->List,P->NoOfWords,sizeof(char *),
        ssSort_Fun);
 qsort((void *)L,NoOfClasses,sizeof(struct AClass),
       csSort_Fun);
 for(i=0,P=L;(i<NoOfCSlots)&&(P->Signature!=NULL);P++,i++)
 {
  if(i)fprintf(Out,"\n");
  for(j=0;*(P->List+j)!=NULL;j++)
  {
   if(j)fprintf(Out," ");
   fprintf(Out,"%s",*(P->List+j));
  }
 }
 fclose(Out);
}
```

Omówienie rozwiązań podanych przez uczniów

Zadanie było sformułowane na tyle jasno, iż nie budziło większych wątpliwości, chociaż trudno jest zidentyfikować, które błędy w rozwiązaniach wyniknęły z niezrozumienia treści zadania, a które były błędami w algorytmach lub w ich implementacjach. Do nieuważnego przeczytania tekstu zadania przyznał się jeden z uczniów, który założył, że w pliku wejściowym jest stała liczba (3000) wyrazów. W niektórych rozwiązaniach nie usuwano powtarzających się wyrazów lub klasy nie były porządkowane.

Znaczna część uczniów podała rozwiązania poprawne pod względem generowanych wyników. Najczęstszą słabą cechą programów był czas obliczeń, przekraczający ustalony limit długości trwania obliczeń dla przykładów testowych – wynikało to z nieumiejętnej lub niestarannej implementacji algorytmów realizujących poszczególne kroki metody.

Zdarzały się błędy w realizacji operacji wejścia/wyjścia (filtru wejściowego), które nieraz ujawniały się w nieoczekiwanych momentach. Przykładem może

być procedura w jednym z rozwiązań, która doklejała wyraz jednoliterowy i następujący po nim znak końca wiersza do wyrazu znajdującego się w kolejnym wierszu słownika.

Wiele trudności stwarzało zarządzanie pamięcią operacyjną – przeważnie wykorzystywano alokację statyczną, co było dużą rozrzutnością. O ile nie było możliwe zarezerwowanie miejsca dla tablicy wystarczającej do przechowywania największych możliwych danych wejściowych (blisko 300MB), o tyle starano się ulokować przynajmniej wskazania do klas w strukturach statycznych. Praktycznie, wszyscy przydzielali wyrazom, niezależnie od ich rzeczywistej długości, 30-znakowe (bajtowe) obszary pamięci. Powoduje to straty przeciętnie w wysokości 50%, a w przypadku krańcowym (tj. słów 1-znakowych) – blisko 100% pamięci operacyjnej.

Kolejnym powodem nieefektywnego wykorzystywania pamięci operacyjnej (i jednocześnie zmniejszenia szybkości obliczeń) było nieusuwanie powtarzających się wyrazów w czasie wczytywania danych wejściowych. W tym przypadku zazwyczaj wykonywano dodatkowe sortowanie list i usuwano z nich powtórzenia dopiero przed wyprowadzeniem wyników.

Sygnaturę wyrazu obliczano zazwyczaj porządkując jego litery. Wykorzystywano do tego sortowanie szybkie (Quicksort) lub rzadziej – algorytm bąbelkowy. Obie te metody w najgorszym przypadku mają kwadratową złożoność czasową. Tylko jeden uczeń zastosował sortowanie kubełkowe, które ma liniową złożoność.

Jeden z uczniów użył jako sygnatury wektora częstości występowania liter w wyrazie. Złożoność czasowa tej metody jest zbliżona do złożoności algorytmu sortowania kubełkowego – wadą jest jednak nieco większa zajętość pamięci. Niektórzy wykorzystali gotowe oprogramowanie z pakietu Turbo Vision, w którym znajduje się między innymi zestaw procedur zarządzających posortowanymi leksykograficznie listami łańcuchów ASCII.

Testy

Aby wychwycić usterki i błędy, które mogły wystąpić w programach uczniów, wykorzystano następujące testy:

— zawierający maksymalnie długą, dopuszczalną, klasę anagramów (3000 wyrazów należących do jednej klasy);
— zawierający maksymalnie dużą, dopuszczalną, liczbę klas (3000 wyrazów, każdy należący do innej klasy);
— zawierający dużą liczbę powtarzających się wyrazów;
— warianty dwóch pierwszych testów dla różnych długości wyrazów (od minimalnej – 1 znak do maksymalnej – 30 znaków);

— warianty powyższych testów zawierające w niektórych wierszach znaki spacji i tabulacji występujące przed i po wyrazie.

Testy te umożliwiły sprawdzenie:

— spełnienia ograniczeń podanych w warunkach zadania;
— efektywności programów;
— poprawnego gospodarowania pamięcią operacyjną;
— odrzucania powtarzających się wyrazów;
— poprawności zastosowanych algorytmów podziału na klasy.

7.3.3. Zadanie NASYCANIE MAKROFANÓW (Autor: Andrzej Walat)

W dalszej treści używamy skróconej nazwy tego zadania MAKROFANY.

Treść zadania MAKROFANY

Makrofany nasycone wytwarza się z ich form pierwotnych – makrofanów nienasyconych – w długim, wieloetapowym procesie produkcyjnym, w którym stopień nasycenia makrofana powiększa się skokowo, co jeden, od 0 do N, gdzie N jest liczbą naturalną dodatnią nie większą niż 144.

Dla każdego stopnia nasycenia $n < N$ istnieje co najmniej jedna i nie więcej niż siedem równoległych ścieżek procesu produkcyjnego, na których stopień nasycenia makrofana podwyższa się od n do $n+1$. Na każdej z tych ścieżek zmienia się temperatura makrofana o pewną całkowitą wartość nie mniejszą niż –3 i nie większą niż 3 stopnie Celsjusza – na każdej ścieżce prowadzącej od stopnia nasycenia n do $n+1$ o inną wartość. Aby zwiększyć o jeden nasycenie makrofana, trzeba wybrać dowolną z nich.

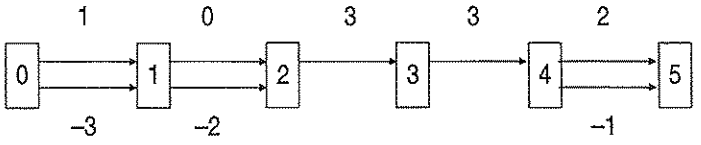
Proces nasycania makrofana może przebiegać różnie, ponieważ możemy wybierać różne ścieżki pomiędzy kolejnymi stopniami nasycenia. Trzeba jednak wiedzieć, że makrofan ginie, gdy jego temperatura spada poniżej 15°C lub przekracza 34°C.

Koszt nasycenia makrofana to suma wartości bezwzględnych wszystkich zmian temperatur jakim podlega na kolejnych ścieżkach procesu nasycania.

Planując przebieg procesu nasycania makrofana dążymy do minimalizacji tego kosztu. Makrofan nie może zginąć.

Przykład

W przypadku opisanym za pomocą następującego grafu, gdzie w kwadratowych polach są wpisane kolejne wartości stopnia nasycenia, a na strzałkach przyrosty temperatur na odpowiednich ścieżkach:



jest osiem różnych możliwości doprowadzenia makrofana nienasyconego o temperaturze początkowej $t_0 = 25$°C do stopnia pełnego nasycenia 5.

W najtańszym przebiegu tego procesu koszt nasycenia wynosi:

$$|1| + |0| + |3| + |3| + |-1| = 8,$$

zaś w najdroższym:

$$|-3| + |-2| + |3| + |3| + |2| = 13.$$

Jeśli makrofan ma temperaturę początkową $t_0 = 33$°C, to istnieje tylko jedna możliwość doprowadzenia go do pełnego nasycenia o koszcie:

$$|-3| + |-2| + |3| + |3| + |-1| = 12.$$

Jeśli makrofan ma temperaturę początkową 34°C, to nie można doprowadzić go do stanu nasycenia tak, aby przeżył.

Zadanie

Ułóż program, który wczytuje z pliku tekstowego MAK.IN zestaw danych o procesie wytwarzania makrofanów nasyconych, a następnie, kolejno dla każdej z dopuszczalnych temperatur początkowych makrofana od 15°C do 34°C, zapisuje w osobnych wierszach pliku MAK.OUT:

- słowo NIE, jeśli nie można doprowadzić makrofana nienasyconego o odpowiedniej temperaturze początkowej do stanu nasycenia tak, aby przeżył, albo
- przebieg procesu nasycania o minimalnym koszcie w postaci ciągu N liczb z zakresu [–3,3] oznaczających zmiany temperatury na kolejnych ścieżkach tego procesu, jeśli taka możliwość istnieje.

Jeśli dla jakiejś temperatury początkowej istnieje wiele przebiegów procesu nasycania makrofana o minimalnym koszcie, to Twój program powinien wypisywać tylko jeden z nich.

Zestaw danych w pliku MAK.IN składa się z $N \le 144$ niepustych wierszy. W każdym z kolejnych wierszy, dla kolejnych stopni nasycenia n od 0 do $N-1$, jest zapisanych nie więcej niż siedem uporządkowanych rosnąco i oddzielonych odstępem liczb całkowitych z zakresu [–3,3] oznaczających odpowiednie przyrosty temperatur na wszystkich ścieżkach zwiększających stopień nasycenia ma-

krofana od n do $n+1$. Bezpośrednio po ostatniej liczbie zapisanej w N-tym wierszu występuje koniec pliku.

Zakładamy, że dane w pliku zostały zapisane bezbłędnie i Twój program nie musi sprawdzać ich poprawności.

Zestaw wyników powinien być zapisany w pliku MAK.OUT w dwudziestu kolejnych wierszach. W każdym wierszu powinno być zapisane słowo NIE albo ciąg N liczb całkowitych z zakresu [–3,3] oddzielonych odstępem. Bezpośrednio po ostatniej liczbie, albo słowie NIE, w dwudziestym wierszu powinien być koniec pliku.

Przykład

Dla pliku MAK.IN:

```
–3 1
–2 0
3
3
–1 2 (koniec pliku)
```

poprawnym zapisem wyników w pliku MAK.OUT jest następujący ciąg dwudziestu wierszy:

```
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 0 3 3 –1
1 –2 3 3 –1
1 –2 3 3 –1
–3 0 3 3 –1
–3 0 3 3 –1
–3 –2 3 3 –1
–3 –2 3 3 –1
NIE (koniec pliku)
```

Na początku programu podaj w komentarzu Twoje nazwisko, imię i numer stanowiska, na którym pracujesz.

Twój program powinien szukać pliku MAK.IN w katalogu bieżącym i tworzyć plik MAK.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę MAK.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku MAK.EXE. Oba te pliki powinny być zapisane na dysku stałym oraz na dyskietce.

Rozwiązanie zadania składa się z programu – tylko jednego – w postaci źródłowej i wykonywalnej, zapisanego zgodnie z podanymi wyżej warunkami na dysku stałym i dyskietce oraz opisu algorytmu wraz z uzasadnieniem jego poprawności.

Rozwiązanie zadania MAKROFANY

Poprawnym i efektywnym rozwiązaniem zadania jest następujący program o nazwie NasycanieMakrofana:

```
program NasycanieMakrofana;
 uses
  Crt;
 const
  MaxN =144;
  Mint = 15;
  Maxt = 34;
  Mindt= -3;
  Maxdt=  3;
 type
  OpisProdukcji=array[0..MaxN-1,Mindt..Maxdt] of Boolean;
  TabelaRuchow =array[0..MaxN-1,Mint..Maxt] of ShortInt;
 var N   :Byte;
     Graf:OpisProdukcji;
     KR  :TabelaRuchow;

 procedure WczytajDane(var N:Byte;var Graf:OpisProdukcji);
  var f  :Text;
      s,t:Byte;
      dt :ShortInt;
 begin
  Assign(f,'MAK.IN');  Reset(f);
  s:=0;
```

```
 repeat
  for dt:=Mindt to Maxdt do Graf[s,dt]:=False;
  repeat
   Read(f,dt);  Graf[s,dt]:=True
  until Eoln(f);
  s:=s+1
 until Eof(f);
 N:=s;
 Close(f)
end; {WczytajDane}

procedure UstalRuchy(var Graf:OpisProdukcji;N:Byte;
                     var KR:TabelaRuchow);
 var MK     :array[0..MaxN,Mint..Maxt] of Integer;
     s,t,nt :Byte;
     MKoszt :Integer;
     Ruch,dt:ShortInt;
begin
 for t:=Mint to Maxt do MK[N,t]:=0;
 for s:=N-1 downto 0 do
  for t:=Mint to Maxt do begin
   MKoszt:=MaxInt;  Ruch:=127;
   for dt:=Mindt to Maxdt do begin
    nt:=t+dt;
    if (Graf[s,dt]) and (nt>=Mint) and (nt<=Maxt)
        and (MK[s+1,nt]<MaxInt)
        and ((Abs(dt)+MK[s+1,nt])<MKoszt) then begin
     MKoszt:=(Abs(dt)+MK[s+1,nt]);
     Ruch:=dt
    end {if}
   end; {for dt:=Mindt}
   MK[s,t]:=MKoszt;  KR[s,t]:=Ruch
  end {for t:=Mint}
end; {UstalRuchy}

procedure WypiszRuchy(var KR:TabelaRuchow;N:Byte);
 var s,tp,tb:Byte;
     dt     :ShortInt;
     f      :Text;
```



```
begin
 Assign(f,'MAK.OUT');  Rewrite(f);
 for tp:=Mint to Maxt do begin
  if KR[0,tp]<127 then begin
   tb:=tp;
   for s:=0 to N-1 do begin
    dt:=KR[s,tb];  tb:=tb+dt;
    Write(f,dt,' ')
   end
  end {if}
  else Write('NIE');
  Writeln(f)
 end; {fot tp:=Mint}
 Close(f)
end; {WypiszRuchy}

begin {Program glowny}
 WczytajDane(N,Graf);
 UstalRuchy(Graf,N,KR);
 WypiszRuchy(KR,N)
end. {NasycanieMakrofana}
```

Omówienie procedury `WczytajDane`

Zadaniem pierwszej procedury o nazwie `WczytajDane` jest wczytanie danych o procesie produkcji i zapisanie ich w dwuwymiarowej tablicy wartości logicznych o nazwie `Graf`. Wykonanie procedury polega na wczytaniu kolejnych wierszy z pliku danych MAK.IN, odpowiadających kolejnym wartościom stopnia nasycenia `s` oraz ustalaniu wartości `Graf[s,dt]=True`, gdy istnieje ścieżka zwiększająca stopień nasycenia makrofana od `s` do `s+1` i zmieniająca jego temperaturę o `dt` i `Graf[s,dt]=False` – w przeciwnym przypadku. Wczytywanie kolejnych wierszy jest połączone z ich liczeniem. W ten sposób ustala się `N`, stopień pełnego nasycenia makrofana.

Omówienie procedury `UstalRuchy`

Głównym elementem rozwiązania jest procedura `UstalRuchy`, wyszukująca od końca ścieżki, którymi należy prowadzić makrofana, aby koszt procesu nasycania był minimalny.

Pomocniczą, ale bardzo istotną rolę odgrywa obliczanie wielkości `MK[s,t]` dla dowolnego `s ∈ [0..N]` oraz `t ∈ [15..34]`, minimalnego kosztu nasycenia makrofana o stopniu nasycenia `s` i temperaturze `t`.

Dla dowolnych wartości stopnia nasycenia `s` oraz temperatury `t` takich, że istnieje możliwość doprowadzenia makrofana o tych parametrach do stanu pełnego nasycenia `N` mamy `MK[s,t] ∈ [0..3*N]`.

Przyjmijmy, że w przypadku, gdy nie istnieje możliwość doprowadzenia makrofana o stanie nasycenia `s` i temperaturze `t` do stanu pełnego nasycenia `N` (bez uśmiercania go) wartość `MK[s,t]` jest również określona i jest nią bardzo duża liczba naturalna `MaxInt`, największa w dopuszczalnym zakresie wartości.

Tablica `MK` jest więc zbiorem wartości dwuargumentowej funkcji o wartościach całkowitych i o argumentach $s \in [0, N]$ oraz $t \in [15, 34]$, spełniającą następujące zależności:

- $MK(N, t) = 0$ dla wszystkich wartości $t \in [15, 34]$,
- dla $s < N$ i dowolnego dopuszczalnego t:

$$MK(s, t) = \min_{t'} \{ | t' - t | + MK(s+1, t') \}$$

po wszystkich t' z dopuszczalnego zakresu temperatur takich, że istnieje ścieżka od stanu nasycenia s do $s+1$ zmieniająca temperaturę makrofana o wartość $dt = t' - t$ oraz $MK(s+1, t') \geq 0$, a jeśli takie t' nie istnieje, to przyjmujemy $MK(s, t)$ równe *MaxInt*.

Te dwie zależności są w gruncie rzeczy rekurencyjną definicją funkcji MK i można je łatwo zamienić na odpowiednią procedurę obliczania jej wartości.

Jednak rekurencyjna procedura obliczania wartości funkcji MK nie byłaby efektywna. Sprowadza ona obliczenie wartości $MK(s, t)$ do obliczania wartości $MK(s', t')$ dla wybranych t' oraz $s' > s$. Te podzadania są ze sobą powiązane, gdyż korzystają z częściowo tych samych wyników kolejnych podzadań. Dlatego rekurencyjna procedura obliczania mnimalnego kosztu nie jest efektywna, ponieważ wielokrotnie wykonuje te same obliczenia.

Z tego powodu w procedurze `UstalRuchy` zastosowano algorytm iteracyjny. Wartości funkcji MK są obliczane od końca, dla s od N do 0 i wszystkie wyniki są zapisywane w dwuwymiarowej tablicy `MK` po to, by można było z nich korzystać wielokrotnie, bez potrzeby powtarzania obliczeń.

Jednocześnie, wraz z ustaleniem minimalnego kosztu `MK[s,t]`, zapisujemy w tablicy `KR` ścieżkę, po której należy wykonać pierwszy ruch od stanu `[s,t]`, w postaci odpowiadającej tej ścieżce wartości `dt`. Jeśli nie ma możliwości doprowadzenia makrofana od stanu `[s,t]` do pełnego nasycenia, to w tablicy `KR` jest zapisywana liczba znacznie przekraczająca zakres dopuszczalnych wahań temperatur, np. `127`, tj. największa dopuszczalna wartość z zakresu `ShortInt`.

Omówienie procedury `WypiszRuchy`

Zadaniem procedury `WypiszRuchy` jest wpisanie do pliku MAK.OUT, dla każdej dopuszczalnej wartości temperatury początkowej `tp`, na podstawie wartości zapisanych w tablicy `KR`, odpowiedniego procesu nasycania makrofana o minimalnym koszcie lub słowa NIE.

Dla dowolnej temperatury początkowej `tp`, jeśli `KR[0,tp]<127`, to istnieje odpowiedni proces o minimalnym koszcie. Pierwszą ścieżkę tego procesu znajdujemy jako wartość `KR[0,tp]`, a następne wyszukujemy w kolejnych kolumnach tej tablicy, zmieniając temperaturę bieżącą makrofana `tb` przez dodanie odpowiedniego przyrostu `dt`.

W przeciwnym przypadku, jeśli `KR[0,tp]=127`, to nie ma możliwości doprowadzenia makrofana do pełnego nasycenia i w odpowiednim wierszu pliku wyjściowego jest wpisywane słowo NIE.

Ocena złożoności rozwiązania

Najwięcej czasu zajmuje wyznaczenie elementów tablic `MK` oraz `KR` – jest on proporcjonalny do iloczynu:

`(Maxt-Mint)*(Maxdt-Mindt)*N`.

Dwa pierwsze czynniki mają stałą wartość, zatem czas obliczeń jest zależny liniowo od N.

Uwagi o metodzie

W przedstawionym rozwiązaniu zastosowano metodę **programowania dynamicznego**. Jest to bardzo ogólna metoda rozwiązywania różnorodnych zadań algorytmicznych, od bardzo prostych po bardzo trudne.

Z reguły zadania te polegają na znalezieniu rozwiązania, które w jakimś sensie ma być optymalne. Drugą charakterystyczną cechą takich zadań jest to, że ich rozwiązanie sprowadza się do rozwiązania podobnych, ale mniejszych i mniej złożonych zadań częściowych, a ostateczny wynik można otrzymać na podstawie rozwiązań zadań częściowych.

Ta druga własność przypomina metodę rozwiązywania znaną jako **dziel i zwyciężaj**. Różnią się te metody tym, że w programowaniu dynamicznym zadania częściowe nie są całkowicie niezależne, gdyż korzystają z wyników otrzymanych dla tych samych podzadań.

Rozwiązując zadanie metodą programowania dynamicznego należy najpierw:

- określić miarę optymalności i zdefiniować odpowiednią funkcję kosztu lub zysku,
- określić, w jaki sposób koszt rozwiązania zależy od kosztów rozwiązań zadań częściowych.

Z reguły, podobnie jak w naszym przypadku, funkcja kosztu ma rekurencyjną definicję.

Aby algorytm rozwiązywania był efektywny, trzeba jednak określić iteracyjny sposób obliczania wartości funkcji kosztu i znajdowania rozwiązań zadań częściowych, połączony z rejestrowaniem wszystkich rozwiązań zadań częściowych, które najczęściej wpisuje się do odpowiednich tablic.

Omówienie rozwiązań podanych przez uczniów

Znikoma liczba uczniów rozwiązywała zadanie omówioną powyżej metodą programowania dynamicznego, a ci którzy jej użyli nie wymienili tej metody z nazwy.

Część uczniów zastosowała następujący algorytm (w jego opisie używamy niektórych zmiennych zadeklarowanych w naszym programie):

Dla każdej temperatury poczatkowej `tp`, kolejno dla każdego `s` od 1 do `N` oraz dla `t` od `Mint` do `Maxt` obliczamy minimalny koszt doprowadzenia makrofana od stanu nasycenia `0` i temperatury `tp`, do stanu nasycenia `s` i temperatury `t`, a wyniki (lub `MaxInt`, jeśli nie istnieje taka możliwość) zapisujemy w dwuwymiarowej tablicy `MK[s,t]`. Jednocześnie ostatni ruch zapisujemy w tablicy `KR[s,t]`. Po wypełnieniu tablic, znajdujemy minimalną wartość `MK[N,t]` dla `t` od `Mint` do `Maxt` i jeśli jest ona mniejsza od `MaxInt`, to na podstawie tablicy `KR` odtwarzamy od końca proces nasycania makrofana o temperaturze początkowej `tp` i o minimalnym koszcie, a w przeciwnym przypadku wpisujemy do pliku wyjściowego słowo NIE.

Jest to rozwiązanie trochę gorsze od podanego przez nas. W tym przypadku czas obliczeń jest proporcjonalny do iloczynu:

`(Maxt-Mint)`^{2}`*(Maxdt-Mindt)*N`.

Ze względu na N jest to jednak nadal zależność liniowa, chociaż współczynnik proporcjonalności jest wielokrotnie większy. Ze względu na zakres dopuszczalnych temperatur zależność jest kwadratowa.

Można sprawdzić, że czas rozwiązywania tego zadania algorytmem polegającym na badaniu wszystkich możliwych przebiegów procesu rośnie wykładniczo ze wzrostem N, a więc taką metodą nie można otrzymać rozwiązania dla dużych N.

Testy

Testy, jak zwykle, dzieliły się na łatwe i trudne.

W tych pierwszych, N było małe, a w konsekwencji liczba wszystkich możliwych przebiegów procesu nasycania makrofana stosunkowo niewielka i każdy poprawny – nawet bardzo nieefektywny – program powinien był znaleźć rozwiązanie w krótkim czasie. Chodziło wyłącznie o sprawdzenie w różnych szczególnych przypadkach czy program znajduje poprawny wynik, zarówno gdy jest on pozytywny, jak i negatywny, i wykrycie różnych możliwych błędów. Na przykład błędu polegającego na tym, że program w każdym kolejnym kroku wybiera ścieżkę o minimalnym koszcie (czyli w sposób zachłanny) lub nie kontroluje, czy makrofan żyje.

Drugą grupę stanowiły testy długie, dla dużego N. Ich głównym celem było rozróżnienie rozwiązań efektywnych od takich rozwiązań, w których czas obliczeń rósł wykładniczo ze wzrostem N. Oczywiście i w tym przypadku chodziło o sprawdzenie, czy program poprawnie znajduje proces nasycania o minimalnym koszcie, nie uśmiercający makrofana albo wpisuje słowo NIE w przypadku przeciwnym.

7.3.4. Zadanie PRZEPUSTOWOŚĆ SIECI (Autor: Maciej M. Sysło)

W dalszej treści używamy skróconej nazwy tego zadania PRZEPUSTOWOŚĆ.

Treść zadania PRZEPUSTOWOŚĆ

Sieć miejska składa się z N $(0 < N \leq 100)$ węzłów ponumerowanych kolejno od 1 do N oraz jednokierunkowych bezpośrednich połączeń pomiędzy węzłami.

Między dowolnymi dwoma węzłami k oraz l istnieją co najwyżej dwa bezpośrednie połączenia: co najwyżej jedno od k do l i co najwyżej jedno od l do k.

Każde bezpośrednie połączenie ma ustaloną przepustowość, będącą liczbą naturalną z przedziału [1, 32000].

Drogą od węzła k do l nazywamy niepusty ciąg różnych bezpośrednich połączeń, taki że:

1. k jest początkiem pierwszego połączenia w tym ciągu,
2. koniec każdego połączenia – prócz ostatniego – jest początkiem następnego połączenia,
3. l jest końcem ostatniego połączenia w tym ciągu.

Przepustowość drogi jest równa najmniejszej przepustowości tworzących ją bezpośrednich połączeń.

Pomiędzy dwoma węzłami sieci może istnieć wiele dróg o różnej przepustowości. Może też nie być żadnej drogi.

Zadanie

Ułóż program, który wczytuje z pliku tekstowego PRZ.IN dane o sieci miejskiej, a następnie dla każdego z przeczytanych z PRZ.IN zapytań w postaci par k l numerów dwóch różnych węzłów wpisuje do kolejnych wierszy pliku tekstowego PRZ.OUT:

- liczbę 0, gdy droga od k do l nie istnieje, albo
- maksymalną przepustowość takiej drogi.

Dane są zapisane w pliku tekstowym PRZ.IN w następującej postaci:

W pierwszym wierszu jest zapisana liczba węzłów N.

Po nim w N kolejnych wierszach występuje N opisów bezpośrednich połączeń wychodzących z kolejnych węzłów sieci.

Po ostatnim takim opisie następuje niepusty ciąg zapytań i koniec pliku.

Opis bezpośrednich połączeń, wychodzących z ustalonego węzła sieci k, jest ciągiem nieujemnych liczb całkowitych, oddzielanych pojedynczymi odstępami. Pierwszy wyraz ciągu jest liczbą tych połączeń lp_k. Jeśli $lp_k = 0$, to na tym ciąg się kończy, a jeśli nie, to dalej występuje lp_k opisów każdego z połączeń w postaci pary liczb. Pierwsza liczba jest numerem węzła, do którego prowadzi połączenie, a druga to jego przepustowość.

Bezpośrednio po ostatniej liczbie każdego opisu bezpośrednich połączeń jest średnik, po którym następuje koniec wiersza.

Każde zapytanie ma postać pary numerów węzłów, tzn. liczb z przedziału [1,100]. Wszystkie zapytania tworzą ciąg, o parzystej liczbie wyrazów, liczb naturalnych oddzielanych odstępem lub pojedynczym znakiem końca wiersza.

Bezpośrednio po ostatniej liczbie występuje koniec pliku.

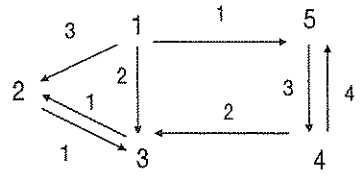
Dane w pliku PRZ.IN są zapisane bezbłędnie i Twój program nie musi tego sprawdzać.

Odpowiedzi powinny być zapisane w pliku PRZ.OUT, w kolejności odpowiadającej kolejności zapytań, w postaci ciągu liczb nieujemnych oddzielanych pojedynczym znakiem końca wiersza. Bezpośrednio po ostatniej odpowiedzi powinien być koniec pliku.

Przykład

Dla pliku PRZ.IN


```
5
3 2 3 3 2 5 1;
1 3 1;
1 2 1;
2 3 2 5 4;
1 4 3;
1 4 4 1 2 3 3 4 1 5 5 2 2 4 4 5 3 4 4 2 (koniec pliku)
```



Twój program powinien utworzyć następujący plik PRZ.OUT:

```
1
0
1
0
1
1
0
4
0
1 (koniec pliku)
```

Na początku programu podaj w komentarzu Twoje nazwisko, imię i numer stanowiska, na którym pracujesz.

Twój program powinien szukać pliku PRZ.IN w katalogu bieżącym i tworzyć plik PRZ.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę PRZ.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku PRZ.EXE. Oba te pliki powinny być zapisane na dysku stałym oraz na dyskietce.

Rozwiązanie zadania składa się z programu – tylko jednego – w postaci źródłowej i wykonywalnej, zapisanego zgodnie z podanymi wyżej warunkami na dysku stałym i dyskietce oraz opisu algorytmu wraz z uzasadnieniem jego poprawności.

Rozwiązanie zadania PRZEPUSTOWOŚĆ

Algorytm Warshalla

Podobnie jak w rozwiązaniu zadania PRZEDSIĘWZIĘCIE, również w tym przypadku podstawową strukturą danych jest **sieć**, czyli **digraf** (tj. graf skierowany) obciążony. Tutaj jednak są obciążone **łuki** odpowiadające połączeniom. W tamtym zadaniu bardzo naturalną i oszczędną reprezentacją sieci były listy,

zrealizowane w przykładowym programie za pomocą typu wskaźnikowego. W rozwiązaniu tego zadania do pamiętania sieci użyjemy **macierzowej reprezentacji** i podamy algorytm, który w istotny sposób wykorzystuje tę reprezentację. W literaturze przedmiotu ten algorytm jest klasyfikowany jako **metoda macierzowa**.

Oznaczmy przez $P = [p_{ij}]$ kwadratową macierz stopnia N, w której element p_{ij} jest równy przepustowości połączenia z i do j w sieci, a więc jest liczbą naturalną z przedziału [1,32000] lub jest równy 0, gdy sieć nie zawiera takiego połączenia. Zauważmy, że treść zadania niejako sugeruje możliwość użycia macierzowej reprezentacji sieci, gdyż liczba węzłów N nie jest większa niż 100, a maksymalne wartości elementów macierzy nie przewyższają 32000, czyli macierz reprezentująca sieć, zadeklarowana jako tablica elementów typu `Integer`, zmieści się w pamięci operacyjnej.

Przyjmijmy, że w sieci połączeń żaden węzeł nie jest połączony sam ze sobą, zatem macierz reprezentująca sieć ma zera na przekątnej.

Zgodnie z przyjętą w treści zadania definicją, jeśli droga z węzła i do węzła j składa się z wielu połączeń, to jej przepustowość jest równa minimalnej przepustowości tych połączeń.

Metoda rozwiązania zadania, którą opiszemy jest realizacją dość ogólnego schematu rozwiązywania problemów optymalizacyjnych, definiowanych na sieciach. Jest ona oparta na dość oczywistym spostrzeżeniu, że droga między dwoma węzłami w sieci jest optymalna (względem ustalonego kryterium), jeśli nie istnieja lepsza droga przechodząca przez być może inne węzły w sieci. W szczególności, dla macierzy danych P oznacza to, że można zwiększyć przepustowość bezpośredniego połączenia między dwoma węzłami i oraz j, jeśli istnieje inny węzeł k o tej własności, że droga z i do j przechodząca przez k ma większą przepustowość. Można to zapisać w postaci następującego stwierdzenia:

Warunek optymalności. *Macierz P zawiera największe przepustowości dróg między każdą parą węzłów wtedy i tylko wtedy, gdy każda trójka węzłów i, j oraz k spełnia tzw.* **warunek trójkąta**:

$$\min(p_{ik}, p_{kj}) \le p_{ij}.$$

Zatem jeśli ten warunek nie jest spełniony, to wykonujemy:

$$\text{jeśli } \min(p_{ik}, p_{kj}) > p_{ij}, \text{ to } p_{ij} := \min(p_{ik}, p_{kj}).$$

Przekonajmy się jeszcze, czy wynik wykonania tej operacji jest poprawny, gdy nie istnieje któreś z występujących w niej bezpośrednich połączeń. Jeśli nie istnieje połączenie z i do k lub z k do j, to odpowiadające tym parom elementy w macierzy P są równe 0, zatem po lewej stronie warunku trójkąta mamy 0 i cały warunek jest trywialnie spełniony. Jeśli natomiast nie istnieje bezpośrednie połączenie z i do j, a istnieją połączenia z i do k oraz z k do j, to warunek trójkąta nie jest spełniony i wykonana zostanie operacja zmiany przepustowości drogi z i do j. Zatem wartość 0 umieszczona w macierzy P, jako przepustowość nie istniejącego połączenia w sieci, gwarantuje poprawne naprawianie warunku trójkąta również w przypadku braku niektórych połączeń (a w ogólności, braku dróg).

Różne metody rozwiązania naszego zadania różnią się kolejnością par węzłów, dla których jest sprawdzany i ewentualnie naprawiany warunek trójkąta.

Z treści zadania wynika, że po wczytaniu danych o sieci powinniśmy umieć określać największe przepustowości dróg między dowolnymi parami węzłów, przy czym nie ma podanych żadnych ograniczeń na liczbę takich par w danych. Możliwe są dwa podejścia:

1. dla kolejnej pary węzłów liczymy największą przepustowość drogi między nimi korzystając z danej macierzy przepustowości bezpośrednich połączeń;
2. najpierw wyznaczamy macierz największych przepustowości dróg między każdą parą węzłów w sieci i na pytanie o największą przepustowość drogi między wybraną parą węzłów sięgamy po wartość odpowiedniego elementu w macierzy.

Treść zadania sugeruje posłużenie się tym drugim podejściem: dane o sieci mogą być zapamiętane w macierzy i nie ma żadnych ograniczeń na liczbę par węzłów, które mogą wystąpić w pytaniach o największe przepustowości dróg. W szczególności mogą się tam znaleźć wszystkie pary różnych węzłów (zobacz poniżej dyskusję o złożoności rozwiązania).

Wybieramy więc to drugie podejście i podamy **algorytm Warshalla**, jedną z najefektywniejszych macierzowych metod rozwiązania, który jest oparty na twierdzeniu Warshalla, podanym oryginalnie dla wyznaczania macierzy przechodniego domknięcia relacji binarnej. Gwarancją poprawności metod macierzowych jest bowiem ich własność, że przeglądają one wszystkie potencjalne drogi w sieci w poszukiwaniu dróg o największych przepustowościach. Liczba dróg w sieci może być jednak bardzo duża – każdy podzbiór węzłów ustawionych w dowolnej kolejności może tworzyć drogę, zatem liczba wszystkich dróg w sieci może przewyższać $N!$, gdzie N jest liczbą węzłów w sieci. Efektywne metody macierzowe zazwyczaj sprawdzają spełnienie warunku trójkąta dla dróg pogrupowanych w pewien sposób. W algorytmie Warszalla drogi są rozpatrywane w zależności od numerów węzłów, przez które przechodzą.

Oznaczmy przez $P^k = [p^k_{ij}]$ macierz największych przepustowości dróg między każdą parą węzłów, których węły pośrednie należą do zbioru {1, 2, ..., k}. Zatem P^0 jest po prostu macierzą P, a P^N jest macierzą największych prze-

pustowości dróg w sieci, czyli szukanym rozwiązaniem. Algorytm Warshalla podaje sposób wyznaczania kolejnych macierzy $P^1, P^2, ..., P^N$ i polega na obliczaniu wartości elementów macierzy P^k na podstawie wartości elementów w poprzedniej macierzy P^{k-1}.

Zauważmy, że na podstawie definicji dróg uwzględnianych w kolejnych macierzach, element w macierzy P^k może zostać zmieniony, jeśli istnieje droga przechodząca przez węzły ze zbioru $\{1, 2, ..., k\}$, której przepustowość jest większa niż przepustowość dróg zawierających jedynie węzły ze zbioru $\{1, 2, ..., k-1\}$. Zatem,

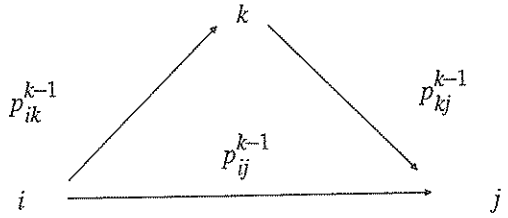
jeśli $\min(p_{ik}^{k-1}, p_{kj}^{k-1}) > p_{ij}^{k-1}$, to przyjmujemy $p_{ij}^{k} := \min(p_{ik}^{k-1}, p_{kj}^{k-1})$.

Możemy zapisać tę operację w postaci (zobacz także jej schemat na rysunku poniżej):

$$p_{ij}^{k} := \max(p_{ij}^{k-1}, \min(p_{ik}^{k-1}, p_{kj}^{k-1})),$$

czyli droga o największej przepustowości z węzła i do węzła j, której wierzchołki pośrednie należą do zbioru $\{1, 2, ..., k\}$ jest drogą, która:

— nie przechodzi przez węzeł k – jej przepustowość jest równa p_{ij}^{k-1}, albo
— zawiera węzeł o numerze k – czyli składa się z dwóch dróg: z i do k o przepustowości p_{ik}^{k-1} oraz z k do j o przepustowości p_{kj}^{k-1}.



Schemat operacji w algorytmie Warshalla

Aby wyznaczyć macierz P^k, należy tę operację wykonać dla wszystkich par węzłów, czyli dla $i = 1, 2, ..., N$, $j = 1, 2, ..., N$. Stąd otrzymujemy następującą postać algorytmu Warshalla:

Algorytm Warshalla

1. Przyjmij $P^0 = P$, gdzie P jest daną macierzą przepustowości bezpośrednich połączeń między węzłami.
2. Dla $k = 1, 2, ..., N$ wykonaj:

$$p_{ij}^{k} := \max(p_{ij}^{k-1}, \min(p_{ik}^{k-1}, p_{kj}^{k-1})),$$

dla $i = 1, 2, ..., N$, $j = 1, 2, ..., N$.

Macierz P^N jest macierzą największych przepustowości dróg między każdą parą węzłów w sieci.

Algorytm Warshalla w powyższej postaci wymaga pamiętania w każdym kroku dwóch macierzy: tworzonej P^k i poprzedniej P^{k-1}. Ale czy rzeczywiście jest konieczne użycie dwóch macierzy? Łatwo można sprawdzić następujące własności operacji naprawiającej warunek trójkąta:

— elementy w k-tym wierszu macierzy P^k są takie same, jak w k-tym wierszu macierzy P^{k-1};
— elementy w k-tej kolumnie macierzy P^k są takie same, jak w k-tej kolumnie macierzy P^{k-1};
— wartość elementu na pozycji ij w macierzy P^k zależy od wartości elementu na tej samej pozycji w macierzy P^{k-1} oraz od wartości elementów należących do k-tych linii (wiersza i kolumny) w tej drugiej macierzy, które nie ulegają zmianie w k-tej iteracji.

Z tych własności wynika, że macierz P^k może być tworzona na miejscu macierzy P^{k-1}, zatem w opisie algorytmu Warshalla powyżej możemy pominąć górne wskaźniki macierzy i ich elementów. W ten sposób stracimy oczywiście informacje o przepustowościach bezpośrednich połączeń w sieci, danych w początkowej macierzy, ale nie są one nam potrzebne. Algorytm Warshalla upraszcza się więc do postaci:

Uproszczony algorytm Warshalla

Wykonaj:

$$p_{ij} := \max(p_{ij}, \min(p_{ik}, p_{kj}))$$

dla $i = 1, 2, ..., N$, $j = 1, 2, ..., N$, $k = 1, 2, ..., N$.

Zanim podamy realizację algorytmu Warshalla w języku Pascal, uzupełnijmy ten algorytm o dodatkową możliwość pamiętania wraz z przepustowościami informacji potrzebnych i wystarczających do znajdowania w krótkim czasie węzłów tworzących optymalne drogi (takie żądanie było umieszczone w pierwotnym sformułowaniu tego zadania olimpijskiego, ale w końcu nie znalazło się w jego ostatecznej postaci). W tym celu użyjemy dodatkowej macierzy $Q = [q_{ij}]$, w której q_{ij} jest numerem węzła bezpośrednio poprzedzającego węzeł j na drodze z i do j o bieżącej przepustowości p_{ij}. Na początku, jeśli istnieje bezpośrednie połączenie z i do j, to przyjmujemy oczywiście $q_{ij} = i$, a jeśli nie

ma takiego połączenia, to przyjmujemy $q_{ij} = 0$. Natomiast w trakcie wykonywania podstawowej operacji algorytmu (zob. rysunek powyżej):

jeśli $p_{ij} < \min(p_{ik}, p_{kj})$, to przyjmujemy $q_{ij} := q_{kj}$.

Procedura WszystkiePrzepustowosci, dla tablicy kwadratowej p o rozmiarach N na N, reprezentujacej sieć połączeń i ich przepustowości, wyznacza w tej samej tablicy największe przepustowości dróg między każdą parą węzłów w sieci oraz tablicę bezpośrednich poprzedników q na drogach o największych przepustowościach. Z kolei, procedura Droga wpisuje do tablicy WezlyDrogi kolejne węzły tworzące drogę o największej przepustowości z i do j – węzły te są pamiętane w tej tablicy na pozycjach od WezlyDrogi[0] do N. (Zauważmy, że drogi o maksymalnych przepustowościach między różnymi węzłami w sieci mogą się składać z różnej liczby połączeń.)

```
const
 Nmax=100;
type
 Wektor =array[0..Nmax] of Integer;
 Macierz=array[1..Nmax,1..Nmax] of Integer;

procedure WszystkiePrzepustowosci(N:Integer;var p,q:Macierz);
 var i,j,k,pp,qq:Integer;
begin
 for i:=1 to N do
  for j:=1 to N do
   if p[i,j]>0 then q[i,j]:=i else q[i,j]:=0;
 for k:=1 to N do
  for i:=1 to N do begin
   pp:=p[i,k];
   if pp>0 then
    for j:=1 to N do begin
     qq:=p[k,j];
     if pp<qq then qq:=pp;
     if p[i,j]<qq then begin
      p[i,j]:=qq;
      q[i,j]:=q[k,j]
     end
    end
  end
end; {WszystkiePrzepustowosci}
```



```
procedure Droga(i,j:Integer; var q:Macierz;
                var WezlyDrogi:Wektor);
 var k,l:Integer;
begin
 WezlyDrogi[N]:=j;
 l:=N;
 k:=j;
 repeat
  k:=q[i,k];
  l:=l-1;
  WezlyDrogi[l]:=k
 until k=i;
 WezlyDrogi[0]:=l
end; {Droga}
```

Pomijamy opis sposobu wczytywania danych i tworzenia na ich podstawie tablicy p – każdy Czytelnik potrafi to uzupełnić samodzielnie.

Złożoność algorytmu Warshalla jest proporcjonalna do N^3, gdyż w każdej z N iteracji jest wykonywanych N^2 operacji, po jednej dla każdego elementu macierzy.

Dysponując macierzą utworzoną przez algorytm Warshalla, odpowiedź na pytanie o największą przepustowość drogi między ustaloną parą węzłów polega na pobraniu wartości odpowiedniego elementu z macierzy P. Zatem, jeśli mamy m takich zapytań, to czas rozwiązania całego zadania jest proporcjonalny do $N^3 + m$. Może być N^2 różnych zapytań, zatem złożoność rozwiązania nie przewyższa nigdy N^3 z niewielkim współczynnikiem.

Inne algorytmy

Inne metody rozwiązywania tego zadania polegają zwykle na odmiennym sposobie przeglądania możliwych dróg między ustaloną parą węzłów w poszukiwaniu drogi o największej przepustowości.

W szczególności można znajdować drogi o największych przepustowościach osobno dla każdej pary węzłów. Taka metody wymaga wykonania przynajmniej N^2 elementarnych operacji dla jednej pary. Zatem jeśli w danych znajduje się N^2 zapytań o największe przepustowości dróg, to całe zadanie zostanie rozwiązane w czasie proporcjonalnym do N^4.

Można jednak zmodyfikować tę ostatnią metodę tak, by jej złożoność była tego samego rzędu, co algorytmu Warshalla. Skorzystamy z dodatkowej własności metody, która liczy największą przepustowość dla ustalonej pary węzłów

k oraz l – najczęściej taka metoda wyznacza największe przepustowości dróg dla wszystkich par węzłów, w których pierwszym węzłem jest k. Zatem dla danej pary k oraz l, jeśli k pojawiło się już w jakiejś parze danych, to odczytujemy wynik z macierzy P, a jeśli nie, to obliczamy największe przepustowości dróg wychodzących z węzła k. W tym ostatnim celu możemy posłużyć się algorytmem o złożoności rzędu N^2 (np. użyć modyfikacji algorytmu Dijkstry), zatem cała metoda będzie miała złożoność proporcjonalną do N^3, czyli jak algorytm Warshalla.

Omówienie rozwiązania podanych przez uczniów

Wszyscy uczniowie odkryli pożytek z warunku trójkąta i proponowali różne sposoby jego naprawiania, jeśli macierz danych go nie spełnia. Wśród dobrych rozwiązań były zarówno takie, w których wykorzystano algorytm Warshalla (przynajmniej trzech uczniów znało ten algorytm i podało jego nazwę), jak i sposoby polegające na znajdowaniu największych przepustowści jedynie dla podanej pary węzłów. W tym drugim przypadku niektórzy zastosowali zmodyfikowany algorytm Dijkstry, a byli i tacy, którzy zauważyli, że wystarczy wywoływać procedurę Dijkstry tylko dla nowych pierwszych węzłów w parach (tak jak opisaliśmy powyżej). Wszystkie te metody korzystały z macierzowej reprezentacji sieci.

Były także próby, ale mniej udane, użycia listowych struktur danych do pamiętania sieci oraz zastosowania rekurencyjnego sposobu przeglądania sieci w poszukiwaniu dróg o największych przepustowościach.

W większości rozwiązań, maksymalne przepustowości dróg były znajdowane niezależnie dla każdej pary z danych wejściowych.

Testy

Test użyte do sprawdzania zadania można naturalnie podzielić na dwie grupy.

Do pierwszej grupy należą takie, w których digraf reprezentujący sieć połączeń miał specjalną strukturę. Na przykład, w teście nr 4 digraf sieci był cyklem. W tym przypadku wszystkie drogi między każdą parą węzłów mają taką samą długość, czyli zawierają taką samą liczbę połączeń równą $N - 1$. W teście nr 5, między wyróżnioną para węzłów istniały drogi o każdej długości (czyli od 1 do $N - 1$), a droga o największej przepustowości składała się z $N - 1$ łuków. Testy te miały na celu sprawdzenie, czy podane przez ucznia rozwiązanie nie ogranicza przeglądania sieci w poszukiwaniu dróg o największej przepustowości do dróg o ustalonej lub ograniczonej długości – a takie rozwiązania były podane.

W pierwszej grupie testów znalazły się również takie sieci, które dla pewnych par węzłów nie zwierają żadnej drogi między nimi – wtedy program rozwiązujący zadanie na pytanie o maksymalną przepustowość drogi między takimi węzłami powinien wpisać wartość 0.

Druga grupa testów miała za zadanie wychwycenie najefektywniejszych 3 rozwiązań, czyli odróżnienie algorytmów o złożoności rzędu N (takich jak algorytm Warszalla) od algorytmów wykonujących większą liczbę działań. Tworzyły ją dwie sieci o odpowiednio 35 i 100 węzłach, w których między każdą parą różnych węzłów istniały bezpośrednie połączenia w obie strony. Zatem grafy tych sieci zawierały odpowiednio 1190 i 9900 łuków i taka była liczba zapytań o największe przepustowości dróg.

Podczas testowania rozwiązań zaobserwowano, zwłaszcza w przypadku użycia testów z tej drugiej grupy, że niektóre programy generowały poprawne odpowiedzi tylko dla niektórych par węzłów. Oznacza to, że zaprogramowane w tych rozwiązaniach algorytmy albo nie uwzględniały wszystkich kombinacji par węzłów, albo nie brały pod uwagę wszystkich możliwych dróg (pod względem długości lub wierzchołków pośrednich) między parami węzłów.

8. SPRAWDZANIE I OCENA ROZWIĄZAŃ ZADAŃ I OLIMIADY INFORMATYCZNEJ

Sprawdzanie rozwiązań

Omawiając zasady sprawdzania i oceny prac zawodników I Olimpiady Informatycznej skupimy najpierw uwagę na zawodach I stopnia. Warto to zrobić z dwu powodów: po pierwsze, jest to ta część zawodów, w której bierze udział największa liczba uczniów, a po drugie – zadania były łatwiejsze niż zadania zawodów II i III stopnia. Można więc omówić metodę sprawdzania rozwiązań, nie wchodząc zbyt głęboko w drugorzędne szczegóły techniczne.

Dla każdej pracy nadesłanej na zawody I stopnia sprawdza się:

— datę stempla pocztowego (czyli nie przekroczenie terminu wysłania rozwiązań),
— kompletność i spełnienie wymagań regulaminu Olimpiady,
— czytelność zapisów programów na dyskietce.

Wszystkie dyskietki są poddawane kontroli antywirusowej. W razie uszkodzenia zapisów podejmuje się próbę ich odtworzenia.

Każda praca otrzymuje numer, pod którym odtąd występuje. Numer pracy wraz z danymi osobowymi ucznia zapisuje się w bazie danych zawodów. Cały proces oceniania jest tak zorganizowany, że jedynym identyfikatorem pracy jest jej numer. Nazwiska uczniów są łączone z listą ocen dopiero po podjęciu wszystkich decyzji o ostatecznych ocenach i zakwalifikowaniu uczniów do zawodów następnego stopnia.

Integralną częścią rozwiązania każdego zadania jest program komputerowy. W I Olimpiadzie wprowadzono komputerowo wspomagane sprawdzanie tych programów. Program sprawdzający pobiera programy uczniów i bada działanie każdego z nich za pomocą ustalonego i jednakowego dla każdego zadania zbioru testów. Dla każdego testu jest podany maksymalny czas działania na nim programu. Ten limit czasu jest tak dobrany, by pozostawić spory margines czasu dla rozwiązań prawidłowych i dostatecznie efektywnych, natomiast wyklucza zbyt długie oczekiwanie na zakończenie działania programu w przypadku jego zapętlenia się lub gdy zastosowana w programie metoda wymusza szczególnie

długie obliczenia (na przykład trwające godzinę lub dłużej), podczas gdy zadanie może być rozwiązane znacznie szybciej (na przykład w ciągu kilku minut).

Dzięki takim testom sprawdza się nie tylko umiejętność napisania programu rozwiązującego zadanie, ale również umiejętność doboru właściwego algorytmu – spośród wielu możliwych metod rozwiązania lepsza jest ta, która szybciej rozwiązuje zadanie dla dużych i złożonych zestawów danych, a przynajmniej nie jest gorsza od innych metod i jej czas działania mieści się w limicie czasu.

Autorami programu sprawdzającego są pracownicy i studenci Instytutu Informatyki Uniwersytetu Warszawskiego: Marcin Kubica, Marek Pawlicki i Krzysztof Stencel. Autorami modułów wspomagających sprawdzanie konkretnych zadań oraz autorami testów, oprócz wyżej wymienionych osób i oprócz autorów zadań, byli mgr Jakub Bojanowski, Marcin Engel, Marcin Jurdziński, Piotr Krysiuk, Marcin Madey, mgr Piotr Filip Sawicki, Tomasz Śmigielski oraz prof. dr hab. inż. Stanisław Waligórski.

Sprawdzanie rozwiązania zadania rozpoczyna się od stwierdzenia, czy program rozwiązujący to zadanie jest na dyskietce i można go uruchomić. Nazwa i sposób zapisania pliku z programem w pamięci komputera są ściśle określone w wymaganiach dotyczących rozwiązań i w treści zadania. Jeśli jest brak programu lub nie można go znaleźć, to autor rozwiązania otrzymuje za nie 0 punktów.

Każdy znaleziony program jest uruchamiany na zestawie testów danego zadania i badane jest dla każdego testu z osobna czy:

— program działa i czas jego działania dla tego testu mieści się w limicie,
— generowany jest plik wyjściowy,
— plik wyjściowy jest niepusty,
— zawartość pliku wyjściowego jest czytelna,
— zawartość pliku wyjściowego jest poprawna merytorycznie, lub zawiera tylko pewne dopuszczalne błędy.

Gdy odpowiedź na którekolwiek z tych pytań jest NIE, to nie są zadawane następne pytania i sprawdzenie dla tego testu kończy się wynikiem negatywnym. Jeśli odpowiedź na wszystkie pytania powyżej jest TAK, to wynik testu jest pozytywny.

W badaniu merytorycznej poprawności rozwiązań pozostawia się pewien margines dla błędów, które są tolerowane. Na przykład jeśli poprawna odpowiedź została w pliku wyjściowym uzupełniona dodatkowym komentarzem (nie pozostawiającym jednak wątpliwości, że po jego usunięciu odpowiedź programu byłaby prawidłowa), to taki plik był traktowany jako dopuszczalny merytorycznie, choć niepoprawny składniowo. Fakt wystąpienia takiej usterki był jednak odnotowywany w wynikach sprawdzenia.

Dla zadania SPONSOR dopuszczalną, ale odnotowaną usterką było użycie typu `Integer` dla zapisania wyniku; treść zadania wskazywała bowiem, że okres obserwacji może być bardzo długi i jeden z testów sprawdzał zdolność programu do zapisywania liczb przekraczających zakres typu `Integer`. W zadaniu PIONKI taką dopuszczalną, ale odnotowaną usterką było znalezienie drogi poprawnej, ale nieoptymalnej.

Punktacja

Punkty za pomyślne przejście testu sprawdzającego metodę rozwiązywania zadania są tak dobierane, aby oceny programów były możliwie najbardziej zróżnicowane: najwięcej punktów jest przyznawanych za test najtrudniejszy, przez który pomyślnie przeszło najmniej rozwiązań, zaś mniej – za test, przez który przeszło więcej programów.

Po sprawdzeniu wszystkich prac oblicza się dla każdego zadania i testu liczbę programów, które przeszły przez ten test z wynikiem pozytywnym. Liczba ta służy dalej jako tzw. współczynnik łatwości testu.

Przyjęto zasadę, że ze 100 punktów, które można było otrzymać za bezbłędne rozwiązanie każdego zadania zawodów I stopnia, 80 punktów otrzymuje się za testy metody. Punkty przyznawane za te testy oblicza się w następujący sposób:

— suma punktów za wszystkie testy metody jednego zadania wynosi 80;
— iloczyn liczby punktów za każdy test metody i współczynnika łatwości tego testu jest dla tego zadania stały.

W razie potrzeby zaokrągla się rezultaty tak, by liczba punktów przyznawana za każdy test była liczbą naturalną, ale bez naruszania warunku 1.

Pozostałe 20 punktów były przydzielane za testy wejścia i wyjścia oraz jako premie za dobrą jakość rozwiązania.

Punktacja rozwiązań zadania TRÓJKĄTY

— 11 testów metody	80 punktów
— za bezbłędne przejście wszystkich tych 11 testów	5 punktów
— 3 testy wejścia i wyjścia po 2 punkty	6 punktów
— za bezbłędne przejście wszystkich tych 3 testów	4 punkty
— za poprawność składniową wszystkich plików z poprawnymi odpowiedziami	5 punktów

Punktacja rozwiązań zadania SPONSOR

— 8 testów metody	80 punktów
— za bezbłędne przejście wszystkich tych 8 testów	5 punktów
— 4 testy wejścia i wyjścia po 2 punkty	8 punktów

— za bezbłędne przejście wszystkich tych 4 testów	2 punkty
— za poprawność składniową wszystkich plików z poprawnymi odpowiedziami	5 punktów

Za bezbłędne przejście testu nr 6 zadania SPONSOR można było otrzymać 29 punktów, jednak w przypadku przekroczenia przez program zakresu liczb typu `Integer`, w którym prawidłowy wynik się nie mieści, można było otrzymać tylko 20 punktów.

Punktacja rozwiązań zadania PIONKI

— 11 testów metody	80 punktów
— za bezbłędne przejście wszystkich tych 11 testów	5 punktów
— 3 testy wejścia i wyjścia po 2 punkty	6 punktów
— za bezbłędne przejście wszystkich tych 3 testów	4 punkty
— za poprawność składniową wszystkich plików z poprawnymi odpowiedziami	5 punktów

W przypadku, gdy dla pewnego testu program znajdował drogę nieoptymalną, liczba punktów przyznawanych za ten test była obniżana do 2/3 punktów przyznawanych za bezbłędną odpowiedź.

Oceny

Wyniki automatycznego sprawdzania programów stanowią podstawę do wyznaczenia liczby punktów przyznanych uczniom za każde zadanie.

Jeśli brak jest programu lub nie można go znaleźć, to uczeń otrzymuje za całe zadanie 0 punktów. Jeśli program źle działa na teście, to za ten test otrzymuje 0 punktów. Pełna ocena całego zadania jest sumą punktów uzyskanych za poszczególne testy oraz, ewentualnie, premii według wykazów powyżej. Dodatnią liczbę punktów za rozwiązanie zadania można uzyskać tylko wtedy, gdy przynajmniej pewne testy zakończyły się wynikiem poozytywnym.

Przydzielanie punktów za testy i obliczanie ocen punktowych za zadania, układanie list numerów prac z ocenami, posortowanych w kolejności nie rosnących liczb punktów uzyskanych za wszystkie zadania jest wykonywane całkowicie automatycznie. Otrzymane listy stanowią następnie podstawę dla decyzji Komitetu Głównego o zakwalifikowaniu uczniów do zawodów następnego stopnia lub przyznaniu nagród. Liczba osób kwalifikowanych jest określona w regulaminie zawodów.

Błędy zawodników i ich skutki

Z opisu sposobów sprawdzania rozwiązań i punktacji wynika, że poza opanowaniem sztuki konstruowania właściwych algorytmów oraz biegłą znajomością

Dla zadania SPONSOR dopuszczalną, ale odnotowaną usterką było użycie typu `Integer` dla zapisania wyniku; treść zadania wskazywała bowiem, że okres obserwacji może być bardzo długi i jeden z testów sprawdzał zdolność programu do zapisywania liczb przekraczających zakres typu `Integer`. W zadaniu PIONKI taką dopuszczalną, ale odnotowaną usterką było znalezienie drogi poprawnej, ale nieoptymalnej.

Punktacja

Punkty za pomyślne przejście testu sprawdzającego metodę rozwiązywania zadania są tak dobierane, aby oceny programów były możliwie najbardziej zróżnicowane: najwięcej punktów jest przyznawanych za test najtrudniejszy, przez który pomyślnie przeszło najmniej rozwiązań, zaś mniej – za test, przez który przeszło więcej programów.

Po sprawdzeniu wszystkich prac oblicza się dla każdego zadania i testu liczbę programów, które przeszły przez ten test z wynikiem pozytywnym. Liczba ta służy dalej jako tzw. współczynnik łatwości testu.

Przyjęto zasadę, że ze 100 punktów, które można było otrzymać za bezbłędne rozwiązanie każdego zadania zawodów I stopnia, 80 punktów otrzymuje się za testy metody. Punkty przyznawane za te testy oblicza się w następujący sposób:

— suma punktów za wszystkie testy metody jednego zadania wynosi 80;
— iloczyn liczby punktów za każdy test metody i współczynnika łatwości tego testu jest dla tego zadania stały.

W razie potrzeby zaokrągla się rezultaty tak, by liczba punktów przyznawana za każdy test była liczbą naturalną, ale bez naruszania warunku 1.

Pozostałe 20 punktów były przydzielane za testy wejścia i wyjścia oraz jako premie za dobrą jakość rozwiązania.

Punktacja rozwiązań zadania TRÓJKĄTY

— 11 testów metody	80 punktów
— za bezbłędne przejście wszystkich tych 11 testów	5 punktów
— 3 testy wejścia i wyjścia po 2 punkty	6 punktów
— za bezbłędne przejście wszystkich tych 3 testów	4 punkty
— za poprawność składniową wszystkich plików z poprawnymi odpowiedziami	5 punktów

Punktacja rozwiązań zadania SPONSOR

— 8 testów metody	80 punktów
— za bezbłędne przejście wszystkich tych 8 testów	5 punktów
— 4 testy wejścia i wyjścia po 2 punkty	8 punktów

— za bezbłędne przejście wszystkich tych 4 testów	2 punkty
— za poprawność składniową wszystkich plików z poprawnymi odpowiedziami	5 punktów

Za bezbłędne przejście testu nr 6 zadania SPONSOR można było otrzymać 29 punktów, jednak w przypadku przekroczenia przez program zakresu liczb typu `Integer`, w którym prawidłowy wynik się nie mieści, można było otrzymać tylko 20 punktów.

Punktacja rozwiązań zadania PIONKI

— 11 testów metody	80 punktów
— za bezbłędne przejście wszystkich tych 11 testów	5 punktów
— 3 testy wejścia i wyjścia po 2 punkty	6 punktów
— za bezbłędne przejście wszystkich tych 3 testów	4 punkty
— za poprawność składniową wszystkich plików z poprawnymi odpowiedziami	5 punktów

W przypadku, gdy dla pewnego testu program znajdował drogę nieoptymalną, liczba punktów przyznawanych za ten test była obniżana do 2/3 punktów przyznawanych za bezbłędną odpowiedź.

Oceny

Wyniki automatycznego sprawdzania programów stanowią podstawę do wyznaczenia liczby punktów przyznanych uczniom za każde zadanie.

Jeśli brak jest programu lub nie można go znaleźć, to uczeń otrzymuje za całe zadanie 0 punktów. Jeśli program źle działa na teście, to za ten test otrzymuje 0 punktów. Pełna ocena całego zadania jest sumą punktów uzyskanych za poszczególne testy oraz, ewentualnie, premii według wykazów powyżej. Dodatnią liczbę punktów za rozwiązanie zadania można uzyskać tylko wtedy, gdy przynajmniej pewne testy zakończyły się wynikiem poozytywnym.

Przydzielanie punktów za testy i obliczanie ocen punktowych za zadania, układanie list numerów prac z ocenami, posortowanych w kolejności nie rosnących liczb punktów uzyskanych za wszystkie zadania jest wykonywane całkowicie automatycznie. Otrzymane listy stanowią następnie podstawę dla decyzji Komitetu Głównego o zakwalifikowaniu uczniów do zawodów następnego stopnia lub przyznaniu nagród. Liczba osób kwalifikowanych jest określona w regulaminie zawodów.

Błędy zawodników i ich skutki

Z opisu sposobów sprawdzania rozwiązań i punktacji wynika, że poza opanowaniem sztuki konstruowania właściwych algorytmów oraz biegłą znajomością

zasad programowania warunkiem sukcesu jest zwracanie uwagi na ścisłe przestrzeganie pewnych reguł postępowania, podawanych w „Zasadach organizacji zawodów" lub w treści zadań.

Otrzymanie zera punktów za zadanie lub za test może być dla ucznia niemiłą niespodzianką, warto więc przyjrzeć się, jakie mogą być tego przyczyny. Oprócz ewidentnych błędów programowania przyczyną niepowodzenia może być zlekceważenie, przez skądinąd niezłego programistę, pewnych zasad, do których może nie być przyzwyczajony i nie zwracać na nie dostatecznej uwagi.

Zadaniem szkoły średniej nie jest kształcenie zawodowych programistów, a raczej uświadomienie młodzieży podstawowych zasad postępowania. W rezultacie często się zdarza, że uczeń, nawet dość dobrze zaznajomiony z systemem plików, nie zwraca większej uwagi na specyficzny dla zadań olimpiady wymóg stosowania ścisłych reguł nazywania plików. W rezultacie, zdarzało się odszukiwać na dyskietkach z rozwiązaniami uczniów pliki z programami ukryte pod najdziwniejszymi nazwami, mającymi mało wspólnego z sugerowanymi w treści zadań.

Sytuacja, gdy program napisany przez ucznia nie jest uruchamiany ręcznie, a przez inny program, który musi go znaleźć wśród wielu innych, uruchomić i sprawdzić jego poprawność, po czym przejść do badania programów innych uczniów, jest jeszcze dla wielu uczniów dość niezwykła. Ze względu na to staraliśmy się w trakcie oceniania prac I Olimpiady podejmować wszelkie starania odszukania programów nawet wtedy, gdy były nazywane niezgodnie z zasadami, nie odejmując za to punktów. W przyszłości jednak takie nieprzestrzeganie obowiązujących reguł będzie powodować obniżanie przyznawanej liczby punktów.

Jeśli program jest i działa, to jednym z powodów otrzymania zera punktów za test, albo za wszystkie testy zadania, może być to, że program nie daje poprawnego pliku wyjściowego: albo go brak, albo plik jest pusty, albo ma nieczytelną zawartość, czyli nie można go ocenić, chociażby nawet był realizacją najlepszego algorytmu z możliwych. Liczy się bowiem skuteczność działania i doprowadzenie dzieła do końca – bez tego wszelkie twory informatyczne są bezużyteczne.

Przyczyną nie znalezienia pliku wyjściowego może być, podobnie jak w przypadku pliku z programem, zlekceważenie wymagań dotyczących zakładania i nazywania plików. W przyszłości takie błędy, nawet jeżeli nie będą się kończyć oceną 0 punktów za test, będą prowadzić do obniżenia ocen.

Inną przyczyną nie działania programu może być nie radzenie sobie z czytaniem pliku wejściowego: w takim przypadku do utworzenia wyników w ogóle nie dochodzi. Może to być spowodowane niezrozumieniem lub złą interpretacją

opisu składni danych wejściowych w treści zadania, a więc błędem nie mającym wiele wspólnego z umiejętnościami programowania lub zakładania plików wyjściowych. Może się także zdarzyć, że program bezskutecznie próbuje czytać dane z pliku, albo czyta je z niewłaściwego, bo wskutek nieuwagi ucznia nazwa pliku, użyta w programie, różni się od tej, pod którą ten plik rzeczywiście występuje w pamięci.

Przyczyną nie utworzenia pliku z wynikami albo przekroczenia limitu czasu może być również próba prowadzenia konwersacji tam, gdzie treść zadania tego nie przewiduje: na przykład żądanie podawania z klawiatury nazwy pliku, która jest jednoznacznie określona w treści zadania, albo żądanie nacisnięcia klawisza ENTER w którejkolwiek fazie obliczeń. Skłonność do takiego postępowania może wynikać stąd, że często w szkołach kładzie się nacisk przede wszystkim na umiejętność pisania programów zdolnych do prawidłowej pracy w trybie konwersacyjnym.

Nieczytelność zawartości pliku wyjściowego może być czasem powodowana pisaniem wyników w postaci niezgodnej z wymaganiami zadania. Może to być tylko próba dodania komentarza albo zmiana układu, ale wprowadzanie takich komplikacji może wskutek błędu programowania skończyć się całkowitym zamazaniem lub zniekształceniem postaci wyników w pliku.

Dopisywanie komentarzy do wyników miewa swoje uzasadnienie: dobrym zwyczajem jest pisać otrzymywane wyniki tak, by od razu było wiadomo, o co chodzi, ich sens był jasny, a układ przejrzysty – chyba że warunki zadania wyraźnie stanowią inaczej, precyzując dokładnie, gdzie i w jakiej formie te wyniki należy zapisywać. Tak było właśnie w zadaniach olimpiady i ci, którzy stosowali się dokładnie do wymagań zadania, dostawali lepsze oceny niż ci, którzy upiększali wyniki dodatkowymi komentarzami.

Zdając sobie sprawę z tego, że przyczyną przydzielania 0 punktów za zadanie lub test mogły być nie tylko błędy programowania czy doboru algorytmu, sprawdzaliśmy szczególnie dokładnie, po zakończeniu sprawdzania komputerowego, właśnie te przypadki. Dodatkowa kontrola przeprowadzana po sprawdzeniu komputerowym, obejmowała także inne przypadki, które wydawały się godne sprawdzenia, jak na przykład wyniki uczniów, którym stosunkowo niewiele brakowało do zakwalifikowania się do zawodów II stopnia.

W pewnych sytuacjach badaliśmy dokładnie, co by było gdyby w programach – tylko na czas badania ich poprawności – usunąć wykryte przez nas błędy lub odstępstwa od reguł podanych w treści zadania. Z zasady stosowaliśmy takie postępowanie w przypadku złożenia przez ucznia reklamacji. Ponieważ było to typowe badanie „co by było gdyby", w żadnym razie nie naruszaliśmy zasady nieingerencji w treść programów w trakcie sprawdzania. Pozwoliło to dokładniej

ocenić skutki błędów i można było naocznie się przekonać o prawdziwości znanej skądinąd w informatyce prawdy, że pewne niedopatrzenia i niedokładności mogą powodować równie złe skutki dla działania i jakości ostatecznego produktu, jak błędy programowania.

Ocenianie prac w zawodach II i III stopnia

Na sposób sprawdzania i oceniania rozwiązań zadań zawodów II i III stopnia w porównaniu z zadaniami zawodów I stopnia wpływ miały między innymi:

— bardziej skomplikowana budowa zawartości plików wyjściowych,
— założenie z treści zadań, że dane wejściowe zawsze spełniają warunki zadania i program rozwiązujący nie musi sprawdzać ich poprawności.

W związku z tym przy sprawdzaniu programów uczniów przyjęto, że:

— wszystkie testy są testami metody,
— nie ma testów wejścia i wyjścia,
— dopuszczono niektóre rozwiązania częściowe,
— premiuje się bezbłędne przejście przez wszystkie testy,
— premiuje się poprawność składniową plików wyjściowych,
— liczba punktów za rozwiązania częściowe zależy od stopnia kompletności rozwiązania.

W zawodach II i III stopnia zastosowano równolegle dwie metody sprawdzania. Pewne testy ze zbioru testów przygotowanych dla każdego zadania były wykonywane w obecności ucznia po zakończeniu zawodów i wyniki tego sprawdzania były zapisywane w formularzu protokołu w jego obecności. Ponadto, sprawdzający wpisywał w protokole swoje uwagi i dodatkowe informacje, na przykład o zastosowanej metodzie rozwiązywania zadania lub o stwierdzonych faktach: braku działającego programu, braku lub wady plików wyjściowych itd.

Niezależnie od tego, programy były sprawdzane automatycznie za pomocą wszystkich testów przygotowanych dla zadań. Sprawdzający prace w obecności ucznia nie znali w tym momencie rezultatów sprawdzania automatycznego. Porównanie wyników obu sprawdzeń wykazywało w każdym przypadku zgodność ocen. Sprawdzanie przy uczniu umożliwia pokazanie mu zasad sprawdzania oraz jest dodatkową kontrolą wyników sprawdzania w przypadkach skrajnych, gdy program nie działał w ogóle, nie dawał żadnych rezultatów w pliku wyjściowym albo gdy wyniki jego działania były całkowicie błędne.

W zawodach II i III stopnia pewna liczba punktów została przeznaczona na zwiększenie oceny szczególnie wyróżniających się rozwiązań.

9. TEKSTY ZADAŃ Z ZAWODÓW MIĘDZYNARODOWYCH

9.1. Zadania Konkursu Informatycznego Krajów Europy Centralnej

9.1.1. Zadanie ZNORMALIZOWANE KWADRATY

Dla potrzeb grafiki komputerowej koduje się (opisuje) części ekranu liczbami całkowitymi dodatnimi zapisanymi z pomocą tylko czterech cyfr 1, 2, 3, 4 (zob. rysunek).

Znormalizowany kwadrat * jest zakodowany liczbą 133, kodem kwadratu + jest 343. Nie ma żadnych ograniczeń wielkości znormalizowanych kwadratów. Chcąc przesunąć kwadrat * na miejsce + można na przykład wykonać 4 przesunięcia w dół i 2 w prawo.

Zadanie

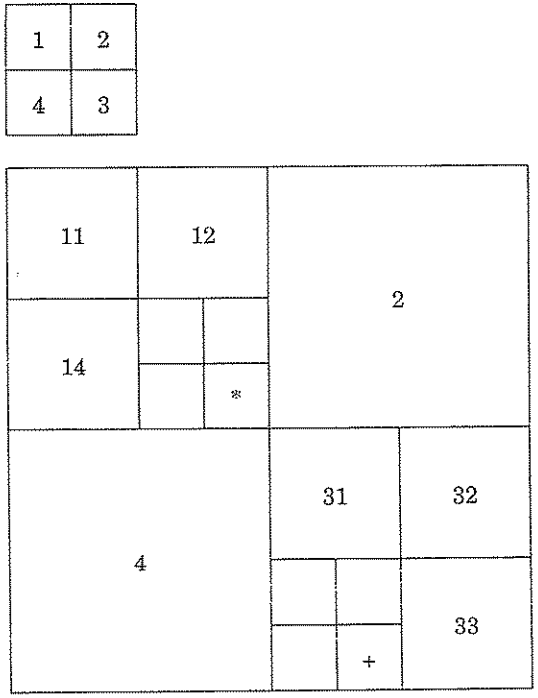
Napisz program (zapisany w pliku SQUARE.PAS), który wykonuje następujące operacje:

a) wczytuje kod znormalizowanego kwadratu zapisany w jednym wierszu pliku danych wejściowych SQUARE.DAT jako liczba całkowita dodatnia, zbudowana wyłącznie z cyfr 1, 2, 3, 4 oraz z następnego wiersza pliku SQUARE.DAT ciąg zbudowany z liter L, R, U, D oznaczający ciąg przesunięć, gdzie L (LEFT) jest przesunięciem w lewo, R(RIGHT) – w prawo, U (UP) – w górę, D (DOWN) – w dół,
b) zapisuje w pliku wynikowym SQUARE.RES końcowe położenie tego znormalizowanego kwadratu albo komunikat OUT OF THE BORDER (w przypadku przesunięcia go poza brzeg),
c) wpisuje do SQUARE.RES pusty wiersz,

powtarzając te operacje, aż do napotkania w pliku SQUARE.DAT pustego wiersza.

Liczba cyfr kodu jest w każdym zestawie danych nie większa niż 35.

Limit czasu: 1 minuta dla każdego zestawu danych.



* ... 133
\+ ... 343
DDDDRR

Przykład

Dla pliku SQUARE.DAT:

133
DDDDRR
1
DD

plik wyników SQUARE.RES powinien zawierać:

343

OUT OF THE BORDER

9.1.2. Zadanie PODZBIORY

Niech n będzie dodatnią liczbą całkowitą. Rozważmy porządek $<$, zwany leksykograficznym, w klasie wszystkich podzbiorów zbioru $\{1, 2, \ldots, n\}$. Niech $S_1 = \{x_1, \ldots, x_i\}$, $S_2 = \{y_1, \ldots, y_j\}$ będą dwoma różnymi podzbiorami $\{1, 2, \ldots, n\}$, gdzie: $x_1 < x_2 < \ldots < x_i$ oraz $y_1 < y_2 < \ldots < y_j$. Mówimy, że $S_1 < S_2$, gdy istnieje takie k, spełniające warunek:

$0 \le k \le \min(i, j)$, że $x_1 = y_1, \ldots, x_k = y_k$ oraz
albo $k = i$, albo $x_{k+1} < y_{k+1}$.

Na przykład podzbiory zbioru $\{1, 2, 3\}$ uporządkowane leksykograficznie tworzą ciąg:

{}	1
{1}	2
{1,2}	3
{1,2,3}	4
{1,3}	5
{2}	6
{2,3}	7
{3}	8

Porządek leksykograficzny, jak pokazuje przykład, przyporządkowuje każdemu podzbiorowi liczbę naturalną.

Twój program powinie wczytywać kolejne wiersze z pliku ASCII o nazwie P6.TXT. Każdy wiersz ma jedną z dwóch postaci:

1 n k

2 n k_1 k_2 ... k_i

Jeśli wiersz ma pierwszą postać, wypisz na ekranie podzbiór zbioru $\{1, 2, \ldots, n\}$ oznaczony liczbą k (przyjmuje się, że $k \le 2^n$).

Jeśli wiersz ma drugą postać, wypisz liczbę oznaczającą podzbiór $\{k_1, \ldots, k_i\}$ zbioru $\{1, \ldots, n\}$ (przyjmując, że $1 \le k_1 < k_2 < \ldots < k_i$).

Twój program powinien dawać wynik w czasie nie dłuższym niż 3 minuty przy założeniu, że $n \le 30$.

9.1.3. Zadanie OBIEKTY

Mamy następujące reguły konstruowania obiektów:
R1. Obiektem początkowym jest kwadrat o boku 1.

R2. Nowy obiekt konstruuje się przez konkatenowanie (łączenie) dwóch obiektów wzdłuż boków jednakowej długości.

R3. Po skonstruowaniu każdego nowego obiektu, przyjmuje się, że dysponujemy nieskończoną liczbą takich obiektów.

Zadania

1. Po wprowadzeniu z klawiatury liczby naturalnej dodatniej n wyświetlić liczbę obiektów o powierzchni co najwyżej n, które można skonstruować, zakładając, że łączy się tylko obiekty o jednakowych rozmiarach.

   Na przykład dla $n = 20$ poprawny wynik ma postać: (1,1) (2,1) (4,2) (8,2) (16,3), gdzie w każdej z par pierwsza liczba jest powierzchnią, a druga – liczbą różnych (nie przystających) obiektów o tej powierzchni.

2) Podobnie, ale przy założeniu, że można łączyć (zgodnie z regułami R1 – R3) obiekty o różnych rozmiarach.

   Na przykład dla $n = 10$ poprawny wynik ma postać: (1,1) (2,1) (3,1) (4,2) (5,1)(6,2)(7,1)(8,2)(9,2)(10,2), gdzie pary oznaczają to samo co w punkcie 1.

Dane wejściowe mają postać dwu liczb:

i n

gdzie $i \in \{1, 2\}$ oznacza wariant zadania, zaś n jest opisaną wyżej liczbą całkowitą.

Wynik jest plikiem tekstowym zawierającym pary elementów opisane wyżej.

Największą możliwą wartością n jest 10000.

Twój program powinien dawać odpowiedź w czasie nie przekraczającym 1 minuty.

9.1.4. Zadanie CZARNE LUB BIAŁE

Napisz program, który wczytuje trzy liczby całkowite dodatnie n, p, q oraz rozstrzyga, czy istnieje ciąg n liczb całkowitych taki, że suma jego dowolnych p kolejnych wyrazów jest dodatnia, a suma dowolnych q kolejnych wyrazów jest ujemna. Jeśli odpowiedź brzmi TAK (YES), to Twój program powinien wypisać taki ciąg.

Wartości n, p, q są wczytywane z klawiatury. Wynikiem jest słowo NO albo słowo YES, po którym następuje wypisany na ekranie ciąg liczb całkowitych.

Przykłady

n=4
p=2
q=3

wynikiem jest

NO

n=6
p=5
q=3

wynik:

YES

–3 5 –3 –3 5 –3

Wartości n są nie większe od 50, limit czasu: 1 minuta.

9.1.5. Zadanie PARZYSTE I NIEPARZYSTE

Rozważmy kraj, w którym jest N miast oraz sieć dróg utworzona przez bezpośrednie połączenia między miastami. Wszystkie bezpośrednie połączenia mają długość 1.

Mówimy, że sieć drogowa jest **parzysto-nieparzysta (even-odd)**, jeśli istnieją dwa miasta, które można połączyć zarówno drogą parzystej długości jak i drogą o nieparzystej długości.

a) Rozstrzygnij, czy sieć drogowa jest parzysto-nieparzysta.

b) Jeśli odpowiedź na a) jest negatywna, wyznacz podzbiór X zbioru miast, o maksymalnej liczbie elementów taki, że dla każdych dwóch miast ze zbioru X, jeśli istnieje pomiędzy nimi droga, ma ona parzystą długość.

Nazwa pliku danych wejściowych jest wprowadzana z klawiatury. W pierwszym wierszu tego pliku podana jest liczba N; kolejne wiersze zawierają pary I, J co oznacza, że istnieje bezpośrednie połączenie pomiędzy miastami I oraz J.

Wartość N jest nie większa niż 300.

Wyniki wyświetl na ekranie w zrozumiałej postaci.

Przykłady

Jeśli plik danych wejściowych zawiera:

5
1 2
2 3
3 4
4 5
5 1

to poprawnym wynikiem jest:

YES

Jeśli plik danych wejściowych zawiera:

3

1 2

poprawnym wynikiem jest:

NO

X ma 2 elementy

X: 2 3

Limit czasu dla każdego pliku danych: 3 minuty.

9.1.6. Zadanie WYRAŻENIA

Wyrażenie zawiera dodawania i mnożenia. Czas potrzebny do wykonania dodawania (+) jest równy p, a czas potrzebny do wykonania mnożenia (*) jest równy q. Czas potrzebny do obliczenia wartości wyrażenia $A \circ B$ jest czasem wykonania operacji $\circ$ plus maksimum z czasów wykonania obliczenia podwyrażenia A i podwyrażenia B (obliczanie wartości podwyrażeń A i B może być wykonywane równolegle i równocześnie).

Argumenty są zmiennymi jednoliterowymi, czas obliczania wartości dowolnego argumentu jest równy 0.

Napisz program, który czyta dane z pliku wejściowego, zawierającego wartości p i q oraz wyrażenia, każde wyrażenie w osobnym wierszu. Stosowanie nawiasów dla wskazania kolejności wykonania operacji jest obowiązkowe. Dla każdego wyrażenia należy:

a) znaleźć i wypisać czas obliczenia wartości tego wyrażenia,

b) znaleźć i wypisać wyrażenie mu równoważne, którego wartość da się obliczyć w możliwie najkrótszym czasie, oraz w następnym wierszu ten najkrótszy czas obliczenia.

Dozwolone są następujące przejścia do wyrażeń równoważnych:

$x+y = y+x;$	$x*y = y*x$	(przemienność),
$x+(y+z) = (x+y)+z;$	$x*(y*z) = (x*y)*z$	(łączność).

Wyniki wpisuje się do pliku wynikowego, w którym pomiędzy danymi dotyczącymi kolejnych wyrażeń wstawia się pusty wiersz.

Na przykład, jeśli plik wejściowy P7.INP zawiera:

1 1

$((a+(b+(c+d)))*e)*f$

$((((a*b)*c)*d)+e)+(f*g)$

to plik wynikowy P7.OUT musi zawierać:

5

$((a+b)+(c+d))*(e*f)$

3

5

$(((a*b)*(c*d))+e)+(f*g)$

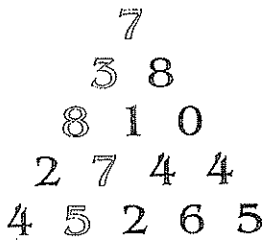
4

9.2. Zadania VI Międzynarodowej Olimpiady Informatycznej

9.2.1. Dzień 1 – Zadanie 1

Rysunek przedstawia trójkąt zbudowany z liczb. Napisz program obliczający największą sumę liczb, przez jakie można przejść po drodze, która zaczyna się w wierzchołku i kończy w pewnym punkcie podstawy.

- Każdy krok można wykonać albo skośnie w lewo w dół, albo skośnie w prawo w dół.
- Liczba wierszy w trójkącie jest >1, ale nie przekracza 100.
- Wszystkie liczby zapisane w trójkącie są całkowite i mieszczą się pomiędzy 0 i 99.



Dane wejściowe

W pierwszej kolejności czyta się z pliku INPUT.TXT liczbę wierszy w trójkącie. W naszym przykładzie INPUT.TXT wygląda tak:

5

7

3 8
8 1 0
2 7 4 4
4 5 2 6 5

Dane wynikowe

Największą sumę zapisuje się, jako liczbę całkowitą, w pliku tekstowym OUTPUT.TXT. W naszym przykładzie:

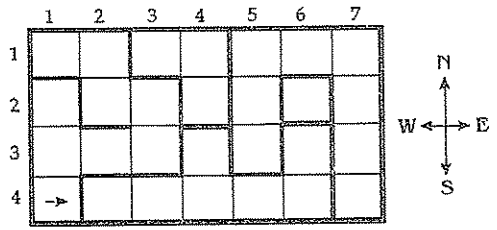
30

9.2.2. Dzień 1 – Zadanie 2

Rysunek pokazuje plan zamku. Napisz program, który oblicza:

1) ile komnat jest w zamku,
2) jak wielka jest największa komnata,
3) którą ścianę należy przebić, aby otrzymać możliwie największą komnatę.

Zamek jest podzielony na $m * n$ (m nie większe niż 50, n nie większe niż 50) kwadratowych modułów. Każdy taki moduł może mieć od zera do czterech ścian.



Strzałka wskazuje ścianę do przebicia zgodnie z wynikami podanymi w przykładzie.

Dane wejściowe

Plan jest zapisany w pliku INPUT.TXT w postaci liczb, po jednej na każdy moduł.

- Na początku pliku są: liczba modułów w kierunku północ-południe oraz liczba modułów w kierunku wschód- zachód.
- W następnych wierszach każdy moduł jest opisany liczbą p (nie mniejszą od 0 i nie większą od 15). Ta liczba jest sumą liczb:

  1 (= ściana na zachód),
  2 (= ściana na północ),
  4 (= ściana na wschód),
  8 (= ściana na południe).

  Ściany wewnętrzne są opisane dwukrotnie; ściana na południe w module 1,1 jest także wymieniona jako ściana na północ w module 2,1.
- Zamek ma zawsze przynajmniej dwie komnaty.

INPUT.TXT dla naszego przykładu:

4
7
11 6 11 6 3 10 6
7 9 6 13 5 15 5
1 10 12 7 13 7 5
13 11 10 8 10 12 13

Dane wynikowe

W pliku OUTPUT.TXT w trzech wierszach pisze się: najpierw liczbę komnat, potem powierzchnię największej komnaty (liczoną w modułach) oraz propozycję, którą ścianę usunąć (najpierw wiersz, potem kolumnę modułu sąsiadującego z tą ścianą i na końcu kierunek kompasu wskazujący na tę ścianę). W naszym przykładzie („4 1 E") jest jedną z wielu możliwości, wystarczy jeśli podasz tylko jedną):

5
9
4 1 E

9.2.3. Dzień 1 – Zadanie 3

Rysunek pokazuje kwadrat. Każdy wiersz, każda kolumna i obie przekątne mogą być odczytywane jako pięciocyfrowe liczby pierwsze. Wiersze czyta się od lewej do prawej. Kolumny czyta się z góry na dół. Obie przekątne odczytuje się od lewej do prawej.

Napisz program konstruujący takie kwadraty na podstawie danych z pliku INPUT.TXT.

- Liczby pierwsze muszą mieć takie same sumy cyfr (w przykładzie: 11).
- Cyfra w lewym górnym rogu kwadratu jest zadana (w przykładzie 1).
- Ta sama liczba pierwsza może wystąpić w kwadracie więcej niż jeden raz.

- Jeśli jest wiele rozwiązań, trzeba pokazać wszystkie.
- Pięciocyfrowa liczba pierwsza nie może się zaczynać od zer, to znaczy 00003 NIE jest pięciocyfrową liczbą pierwszą.

1	1	3	5	1
3	3	2	0	3
3	0	3	2	3
1	4	0	3	3
3	3	3	1	1

Dane wejściowe

Program wczytuje dane z pliku INPUT.TXT, najpierw sumę cyfr liczb pierwszych, następnie cyfrę w lewym górnym rogu kwadratu. Plik zawiera dwa wiersze. Dla danych testowych rozwiązanie będzie zawsze istniało.

W naszym przykładzie danymi wejściowymi są:

11
1

Dane wynikowe

W pliku OUTPUT.TXT zapisz pięć wierszy dla każdego znalezionego rozwiązania, gdzie każdy wiersz jest z kolei pięciocyfrową liczbą pierwszą. Przykład jak wyżej ma 3 rozwiązania, co oznacza, że plik OUTPUT.TXT zawiera następującą treść (pusty wiersz pomiędzy różnymi rozwiązaniami nie jest konieczny):

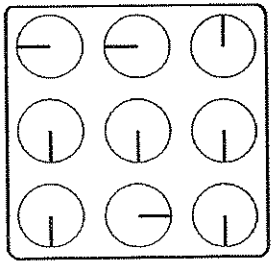
11351
14033
30323
53201
13313

11351
33203
30323
14033
33311

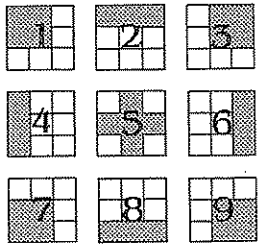
13313
13043
32303
50231
13331

9.2.4. Dzień 2 – Zadanie 1

W tablicy 3*3 jest 9 zegarów (rys. 1). Chodzi o ustawienie tarcz wszystkich zegarów ponownie w pozycji godziny 12, w możliwie najmniejszej liczbie ruchów. Jest dziewięć różnych dopuszczalnych sposobów obracania tarcz zegarów. Ruchem jest zastosowanie dowolnego z tych sposobów. W każdym ruchu wybiera się jedną liczbę od 1 do 9 włącznie: wybranie tej liczby pociąga za sobą obrócenie o 90 stopni w kierunku ruchu wskazówek zegara tarcz tych zegarów, które są oznaczone szarym kolorem na rysunku 2.



Rysunek 1



Rysunek 2

Dane wejściowe

Wczytaj dziewięć liczb z pliku wejściowego INPUT.TXT. Te liczby dają położenia początkowe tarcz zegarów:

0 = godzina 12,
1 = godzina 3,
2 = godzina 6,
3 = godzina 9.

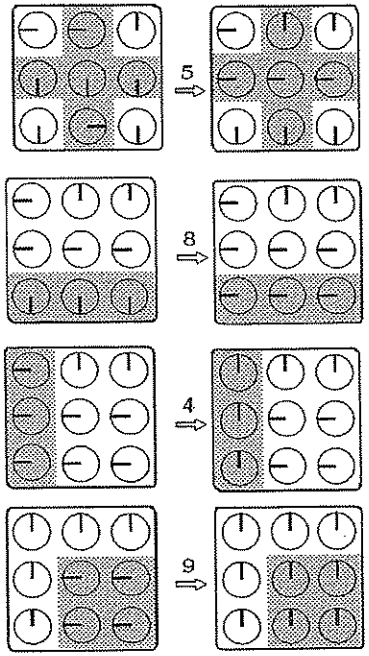
Przykład na rys. 1 daje następującą zawartość pliku wejściowego:

3 3 0
2 2 2
2 1 2

Dane wynikowe

Wpisz do pliku OUTPUT.TXT możliwie najkrótszą sekwencję ruchów, która obraca wszystkie tarcze zegarów do położenia godziny 12. Jeśli jest wiele rozwiązań, należy podać tylko jedno. W naszym przykładzie plik OUTPUT.TXT może wyglądać tak:

5849



Rysunek 3

9.2.5. Dzień 2 – Zadanie 2

Człowiek przychodzi na przystanek autobusowy o 12.00 i pozostaje tam w czasie od 12.00 do 12.59. Na przystanku zatrzymują się autobusy wielu linii autobusowych. Człowiek notuje czasy przyjazdu autobusów. Czasy przyjazdu autobusów są ustalone i autobusy przyjeżdżają według rozkładu.

- Autobusy tej samej linii przyjeżdżają w ciągu całej godziny pomiędzy 12.00 a 12.59 w regularnych odstępach.

- Czasy podaje się w całych minutach od 0 do 59.
- Autobusy każdej linii zatrzymują się na tym przystanku (i w tym czasie) przynajmniej dwa razy.
- W testach liczba linii autobusowych nie przekroczy 17.
- Autobusy różnych linii mogą przyjeżdżać w tym samym czasie.
- Wiele linii autobusowych może mieć te same czasy pierwszego przyjazdu i/lub te same odstępy między kolejnymi autobusami. Jeśli dwie linie autobusowe mają ten sam czas pierwszego przyjazdu i ten sam odstęp między kolejnymi autobusami, to są różne i należy wymienić je obie.

Ustal rozkład jazdy z najmniejszą możliwą liczbą linii autobusowych zatrzymujących się na przystanku, tak aby ten rozkład był zgodny z danymi wejściowymi. Dla każdej linii autobusowej podaj czas pierwszego przyjazdu i odstęp między kolejnymi autobusami.

Dane wejściowe

Plik wejściowy INPUT.TXT zawiera liczbę n (nie przekraczającą 300) zanotowanych przyjazdów autobusów, po której następują czasy przyjazdów autobusów w porządku niemalejącym.

W naszym przykładzie:

17

0 3 5 13 13 15 21 26 27 29 37 39 39 45 51 52 53

Dane wynikowe

Wpisz do pliku OUTPUT.TXT po jednym wierszu dla każdej linii autobusowej. Każdy wiersz w pliku podaje *czas przyjazdu pierwszego autobusu* i *odstęp między kolejnymi autobusami w minutach*. Kolejność linii autobusowych nie ma znaczenia. Jeśli jest wiele rozwiązań, trzeba podać tylko jedno z nich.

W naszym przykładzie:

0 13

3 12

5 8

9.2.6. Dzień 2 – Zadanie 3

Masz koło podzielone na sektory (wycinki).

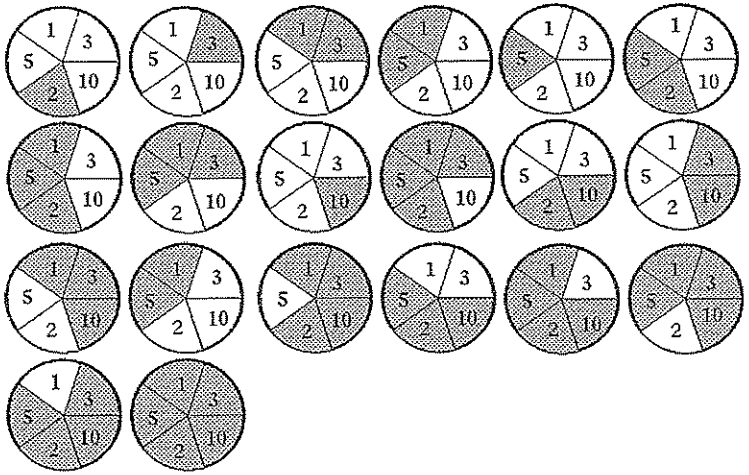
Masz trzy liczby: k (nie przekraczająca 20), n (nie przekraczająca 6) oraz m (nie przekraczająca 20). n jest liczbą sektorów. Wybierz liczby całkowite, które powinny być wpisane po jednej w każdym z sektorów. Wszystkie liczby powinny być większe lub równe k.

Gdy wszystkie sektory są zapełnione, możesz brać liczbę całkowitą z sektora lub utworzyć nową liczbę sumując liczby z dwu lub więcej sąsiadujących sektorów.

Mając te liczby, możesz utworzyć nieprzerwany ciąg wszystkich liczb całkowitych pomiędzy m oraz pewną liczbą i (m, m+1, m+2, ..., i).

Zadanie polega na wybraniu całkowitych liczb wpisywanych do sektorów tak, by największa liczba w tym ciągu (i) była możliwie największa.

Rysunek pokazuje, jak należy generować wszystkie liczby od 2 do 21. Szare pola oznaczają sektory, z których bierze się liczby do sumowania.



Dane wejściowe

Plik INPUT.TXT zawiera trzy liczby (n, m oraz k).

Przykład:


```
5
2
1
```

Dane wynikowe

Plik OUTPUT.TXT musi zawierać:

- Największą liczbę (i), którą można wygenerować mając ustaloną listę liczb (wpisanych do kolejnych sektorów).
- Wszystkie ustawienia liczb w kole, które pozwalają utworzyć ciąg od m do i (po jednym na każdy wiersz). Każde ustawienie jest listą liczb, zaczynającą się od liczby najmniejszej (która nie koniecznie musi być jedyna).

(2 10 3 1 5) NIE jest poprawnym rozwiązaniem zadania, ponieważ nie zaczyna się od najmniejszej liczby. (1 3 10 2 5) i (1 5 2 10 3) muszą być oba w pliku wynikowym. Zauważ, że (1 1 2 3), (1 3 2 1), (1 2 3 1) oraz (1 1 3 2) powinny wszystkie być w pliku wynikowym.

Przykład

```
21
1 3 10 2 5
1 5 2 10 3
2 4 9 3 5
2 5 3 9 4
```