Jakub Radoszewski

MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

II OLIMPIADA INFORMATYCZNA

1994/1995

WARSZAWA, WROCŁAW – 1995

Autorzy:

Rozdz. 1: prof. dr hab. Maciej M. Sysło
Rozdz. 2–5: Komitet Główny Olimpiady
Rozdz. 6: prof. dr hab. inż. Stanisław Waligórski
Rozdz. 7: Autorzy zadań: dr Piotr Chrząstowski-Wachtel, dr Krzysztof Diks, Marcin Jurdziński, Piotr Krysiuk, prof. dr hab. Wojciech Rytter, Tomasz Śmigielski, dr Andrzej Walat
Rozdz. 8: mgr Marcin Kubica
Rozdz. 9: mgr Krzysztof Stencel, 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

ISBN 83–902611–2–X

Nakład 2000 egz.

SPIS TREŚCI

1. WSTĘP

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

W grudniu 1993 roku, po spełnieniu wymogów zarządzenia Ministra Edukacji Narodowej w sprawie organizacji olimpiad przedmiotowych, Instytut Informatyki Uniwersytetu Wrocławskiego powołał Olimpiadę Informatyczną.

W roku szkolnym 1993/1994 odbyła się już II Olimpiada Informatyczna. Oddajemy do rąk Czytelników szczegółową relację z jej przebiegu oraz z zawodów międzynarodowych, w których brali udział zdobywcy czterech pierwszych miejsc w olimpiadzie krajowej.

Ponieważ informacje o Olimpiadzie Informatycznej nie dotarły jeszcze do wszystkich szkół, zawarliśmy w tym opracowaniu również oficjalne dokumenty Komitetu Głównego: Regulamin Olimpiady Informatycznej i Zasady organizacji zawodów w 1994/1995 roku. Te dwa dokumenty zawierają wiele informacji ważnych dla uczestników olimpiad a dotyczących organizacji zawodów, formalnych wymogów stawianych rozwiązaniom zadań oraz wyróżnień i nagród przyznawanych uczniom i nauczycielom.

Nowością II Olimpiady było przeprowadzenie zawodów II stopnia w trzech okręgach, w Toruniu, Warszawie i we Wrocławiu. Dzięki temu mogła być zwiększona, z 40 do 100 uczniów, liczba uczestników zawodów II stopnia.

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 formalnych 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 efektywanie dzięki użyciu właściwego algorytmu.

O tym, jak dobierać algorytmy, piszą autorzy zadań, starając się zwrócić uwagę na podstawowe czynniki decydujące o jakości rozwiązań. Powinno to skłonić zainteresowanych uczniów do dalszego pogłębiania swojej wiedzy i umiejętności oraz sięgnięcia po podręczniki poświęcone algorytmice, algorytmom i strukturom danych i metodom obliczeniowym z różnych dziedzin. Listę polecanych książek zamieszczamy na końcu tych materiałów.

W materiałach są zamieszczone również relacje z dwóch międzynarodowych olimpiad informatycznych (rozdz. 6), w których brała udział ekipa Polski. Rozdział 9 zawiera teksty zadań z zawodów międzynarodowych. Zwracamy uwagę na pojawienie się w Olimpiadzie Międzynarodowej dwóch nowych rodzajów zadań. Jedno – wymagało podania rozwiązania w postaci programu interakcyjnego, a drugie – polegało na udzieleniu pisemnej odpowiedzi, bez użycia komputera. Takie zadania mogą pojawić sie wkrótce na krajowej olimpiadzie informatycznej.

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

Uwaga. W omówieniu rozwiązań zadań w rozdziale 7, autorzy odwołują się do materiałów I Olimpiady Informatycznej, które w dalszej treści są oznaczone przez *I-OI*[*].

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

2. REGULAMIN OLIMPIADY INFORMATYCZNEJ[*]

§ 1. Wstęp

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

§ 2. Cele Olimpiady Informatycznej

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

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

§ 3. Organizacja Olimpiady

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

§ 4. Komitet Główny Olimpiady Informatycznej

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

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

§ 5. Komitety Okręgowe

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

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

§ 6. Przebieg Olimpiady

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

§ 7. Uprawnienia i nagrody

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

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

§ 8. Finansowanie Olimpiady

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

§ 9. Przepisy końcowe

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

3. ZASADY ORGANIZACJI ZAWODÓW W ROKU SZKOLNYM 1994/95

Podstawowym aktem prawnym dotyczącym Olimpiady jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach oświaty*. „Zasady organizacji zawodów" są uzupełnieniem tego regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 1994/95.

§ 1. Wstęp

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

§ 2. Organizacja Olimpiady

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

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

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

§ 3. Wymagania dotyczące rozwiązań zadań zawodów I stopnia

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

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

w nieprzekraczalnym terminie nadania do poniedziałku 14.11.1994 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki. Rozwiązania dostarczane w inny sposób nie będą przyjmowane.
Rozwiązanie wszystkich zadań nie jest warunkiem udziału w Olimpiadzie.
2. Rozwiązanie każdego zadania składa się z:
   a) programu (tylko jednego) na dyskietce w postaci źródłowej i skompilowanej (w Logo – tylko w postaci źródłowej),
   b) wydrukowanego tekstu tego programu w postaci źródłowej,
   c) opisu algorytmu rozwiązania zadania z uzasadnieniem jego poprawności.
3. 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 (w Logo – tylko w postaci źródłowej),
   — spis zawartości dyskietki w pliku nazwanym SPIS.TRC.

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

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

6. Opcje kompilatora powinny być częścią tekstu programu. Zaleca się stosowanie opcji standardowych.
7. Prace niesamodzielne lub zbiorowe nie będą brane pod uwagę.

§ 4. Uprawnienia i nagrody

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

5. Komitet Główny ustala skład reprezentacji Polski na VII Międzynarodową Olimpiadę Informatyczną w 1995 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na VII Międzynarodową Olimpiadę Informatyczną, przed rozpoczęciem zawodów III stopnia.
6. Nauczyciel (opiekun naukowy), który przygotował laureata Olimpiady Informatycznej, otrzymuje nagrodę przyznawaną przez Komitet Główny Olimpiady.
7. Uczestnicy zawodów II i III stopnia otrzymują nagrody rzeczowe.

§ 5. Przepisy końcowe

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

Uwaga. W materiałach rozsyłanych do szkół, po „Zasadach organizacji zawodów" zostały umieszczone treści trzech zadań zawodów I stopnia, a po nich – następujące sugestie dla rozwiązujących zadania:

1. Przeczytaj uważnie nie tylko tekst zadań, ale i treść „Zasad organizacji zawodów".
2. Przestrzegaj dokładnie warunków określonych w tekście zadania, w szczególności wszystkich reguł dotyczących nazw plików.
3. Staraj się dobrać taką metodę rozwiązania zadania, która jest nie tylko poprawna, ale daje wyniki w jak najkrótszym czasie.
4. Ocena za rozwiązanie zadania jest określana na podstawie wyników testowania programu i uwzględnia poprawność oraz efektywność metody rozwiązania użytej w programie.

4. REGULAMIN PRZEPROWADZENIA ZAWODÓW II i III STOPNIA

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

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

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

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

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

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

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

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

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

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

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

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

5. SPRAWOZDANIE Z PRZEBIEGU II 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 *I-OI*, rozdz. 2).

5.1. Organizacja zawodów

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

Dnia 15 października 1994 roku rozesłano plakaty zawierające zasady organizacji zawodów I stopnia (zob. rozdz. 3) oraz zestaw 4 zadań konkursowych do 2916 szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 24 października 1994 roku. Ostatecznym terminem nadsyłania prac konkursowych był 14 listopada 1994 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w trzech okręgach w dniach 9 – 11.02.1995 roku, natomiast zawody III stopnia odbyły się w siedzibie Komitetu Głównego (OEIiZK) w Warszawie w dniach 3 – 5.04.1995 roku.

Uroczystość zakończenia II Olimpiady Informatycznej odbyła się w dniu 7 kwietnia 1995 roku w sali posiedzeń Rady Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego przy ul. Banacha 2.

5.2. Skład osobowy Komitetów Olimpiady Informatycznej

5.2.1. Komitet Główny

Prezydium:

przewodniczący

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

zastępca przewodniczącego

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

sekretarz naukowy

dr Andrzej Walat, OEIiZK,

kierownik organizacyjny

Tadeusz Kuran, OEIiZK,

sekretarz

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

Członkowie:

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

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

Komitet Główny odbył 9 posiedzeń, a Prezydium – 3; protokoły z tych posiedzeń znajdują się w sekretariacie Olimpiady.

5.2.2. Komitety Okręgowe

Komitet Okręgowy w Warszawie

przewodniczący:

dr Krzysztof Diks, Uniwersytet Warszawski,

członkowie:

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

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

Komitet Okręgowy we Wrocławiu

przewodniczący:

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

zastępca przewodniczącego:

dr Krzysztof Loryś, Uniwersytet Wrocławski,

sekretarz:

inż. Maria Woźniak, Uniwersytet Wrocławski,

członkowie:

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

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

Komitet Okręgowy w Toruniu

przewodniczący:

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

sekretarz:

dr Bolesław Wojdyło, Uniwersytet Mikołaja Kopernika w Toruniu,

członkowie:

mgr Anna Kwiatkowska, IV Liceum Ogólnokształcące w Toruniu,
dr Krzysztof Skowronek, Kuratorium Oświaty w Toruniu,
dr Mirosława Skowrońska, Uniwersytet Mikołaja Kopernika w Toruniu.

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

5.3. Jury

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

mgr Marcin Engel	– doktorant,
Marcin Jurdziński	– student,
Piotr Krysiuk	– student,
mgr Marcin Kubica	– doktorant,
mgr Adam Malinowski	– pracownik (doktorant),
Marcin Madey	– student,
mgr Wojciech Plandowski	– pracownik (doktorant),
Marek Pawlicki	– student,
mgr Piotr F. Sawicki	– pracownik (doktorant),
mgr Krzysztof Stencel	– pracownik (doktorant),
Tomasz Śmigielski	– student.

5.4. Zawody I stopnia

W zawodach I stopnia II Olimpiady wzięło udział 508 uczniów, którzy nadesłali łącznie 1665 rozwiązań zadań, w tym:

432 rozwiązania zadania DRZEWA,

422 rozwiązania zadania JEDYNKI I ZERA,

350 rozwiązań zadania OPTYMALIZACJA DYSKU,

461 rozwiązań zadań PALINDROMY.

Jury podjęło decyzję o sprawdzeniu 39 rozwiązań zadań przysłanych przez uczniów w kopiach zapasowych:

10 rozwiązań zadania DRZEWA,

9 rozwiązań zadania JEDYNKI I ZERA,

10 rozwiązań zadania OPTYMALIZACJA DYSKU,

10 rozwiązań zadania PALINDROMY.

Z rozwiązaniami

czterech zadań nadeszły	– 283 prace,
trzech zadań	– 118 prac,
dwóch zadań	– 72 prace,
jednego zadania	– 35 prac.

Przysłano 13 dyskietek, które były częściowo lub całkowicie nieczytelne, z czego na 11 udało się odzyskać zapisy za pomocą programów Norton Utility. Nie udało się odzyskać informacji z dysku zawodnika nr 147. Na 15 dyskietkach wykryto i usunięto różnego rodzaju wirusy.

Uczestnicy I etapu reprezentowali 46 województw. Nie było uczniów z województw: konińskiego (w I Olimpiadzie również), przemyskiego i tarnobrzeskiego.

Najliczniej były reprezentowane następujące województwa:

st. warszawskie	– 74 uczniów,	łódzkie	– 18,
katowickie	– 43,	poznańskie	– 18,
gdańskie	– 30,	bydgoskie	– 17,
krakowskie	– 24,	wałbrzyskie	– 17,
szczecińskie	– 24,	lubelskie	– 16,
tarnowskie	– 20,	toruńskie	– 16,
rzeszowskie	– 20,	kieleckie	– 15,
wrocławskie	– 19,	częstochowskie	– 10.

W zawodach I stopnia najliczniej były reprezentowane szkoły:

1. IV L.O. im. T. Kościuszki z Torunia	– 15 uczniów,
2. III L.O. im. Marynarki Wojennej RP z Gdyni	– 12,
3. II L.O. im. St. Batorego z Warszawy	– 10,
4. I L.O. im. S. Konarskiego z Mielca	– 9,
5. XXVII L.O. im. T. Czackiego z Warszawy	– 9,
6. IV L.O. im. St. Anioła z Tarnowa	– 8,
7. V L.O. im. A. Witkowskiego z Krakowa	– 7,
8. XIV L.O. im. Polonii Belgijskiej z Wrocławia	– 7,
9. VI L.O. im. J. i J. Śniadeckich z Bydgoszczy	– 6,
10. VIII L.O. im. A. Mickiewicza z Poznania	– 6,
11. XIII L.O. ze Szczecina	– 6,
12. L.O. im. Wł. Jagiełły z Dębicy	– 5,
13. XIX L.O. ze Szczecina	– 5,
14. XIV L.O. im. S. Staszica z Warszawy	– 5,
15. V L.O. z Bielska Białej	– 4,
16. II L.O im. R. Traugutta z Częstochowy	– 4,
17. Zespół Szkół Łączności z Krakowa	– 4,
18. VIII L.O. im. A. Asnyka z Łodzi	– 4,
19. XXXI L.O. im. L. Zamenhofa z Łodzi	– 4,
20. II L.O. im. H. Kołłątaja z Wałbrzycha	– 4,
21. IX L.O. im. K. Hoffmanowej z Warszawy	– 4,
22. Technikum Elektroniczno-Mechaniczne z Warszawy	– 4,
23. V L.O. im. J. Poniatowskiego z Warszawy	– 4.

Do zawodów I stopnia decyzją Komitetu Głównego Olimpiady dopuszczono 3 uczniów ze szkół podstawowych: nr 143 i nr 337 z Warszawy oraz nr 19 z Krakowa.

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

Warszawa	– 63 uczniów,	Mielec	– 11 uczniów,
Szczecin	– 21,	Wałbrzych	– 11,
Kraków	– 20,	Częstochowa	– 9,
Wrocław	– 17,	Tarnów	– 9,
Łódź	– 16,	Bydgoszcz	– 8,
Toruń	– 16,	Lublin	– 8,
Gdynia	– 15,	Gliwice	– 7,
Poznań	– 13,	Kielce	– 7,
Gdańsk	– 11,	Ostrowiec Św.	– 7.

416 uczniów podało klasę, do której uczęszczali. W tym:

do I klasy szkoły średniej	– 21 uczniów,
do II klasy szkoły średniej	– 92,
do III klasy szkoły średniej	– 139,
do IV klasy szkoły średniej	– 138,
do V klasy szkoły średniej	– 23,
do VIII klasy szkoły podstawowej	– 3.

Największa liczba uczniów zadeklarowała użycie języka programowania:

Turbo Pascal (Borland)	– 410 uczniów,
Borland C/C++	– 49,
Basic	– 20.

Ponadto posługiwano się językami:

Clipper 5.01, 5.2, 5.2d	– 3 uczniów,
Lattic C 6.5	– 1 uczeń,
HighSpeed Pascal 1.5	– 4 uczniów,
Maxon C++ 1.11	– 1 uczeń,
Maxon Pascal 1.5	– 1,
Microsoft Visual C++ 1.0 Prof.Edition	– 1,
Personal C 1.2b	– 1,
SAS/C Amiga Compiler 6.50	– 1,
TC++ 1.01	– 1,
TC	– 1,
Visual Basic 3.0	– 1.

Komputerowe wspomaganie umożliwiło sprawdzenie prac z zawodów I stopnia kompletem 61 testów w ciągu kilkunastu godzin.

Nie udało się sprawdzić pracy jednego z uczniów, ponieważ nie przysłał żadnej dokumentacji swojej pracy, co uniemożliwiło ustalenie parametrów modułów użytych przez niego w trakcie pisania programu. Komitet Główny podjął decyzję o wykluczeniu z zawodów dwóch uczniów, którzy nadesłali identyczne rozwiązania jednego z zadań.

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

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

Tabela 1.

Liczba punktów	DRZEWA		JEDYNKI I ZERA		OPTYMALIZACJA DYSKU		PALINDROMY	
	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo
100	87	20,1%	45	10,7%	36	10,3%	77	16,7%
od 99 do 75	75	17,4%	27	6,4%	21	6,0%	129	28,0%
od 74 do 50	62	14,4%	43	10,2%	48	13,7%	154	33,4%
od 49 do 1	141	32,6%	276	65,4%	224	64,0%	84	18,2%
0	67	15,5%	31	7,3%	21	6,0%	17	3,7%

Wyniki za wszystkie 4 zadania są zawarte w Tabeli 2.

Tabela 2.

SUMA	liczba uczniów	procentowo
400 pkt.	8	1,6%
od 399 do 300 pkt.	56	11,0%
od 299 do 200 pkt.	124	24,5%
od 199 do 1 pkt.	300	59,2%
0 pkt.	19	3,7%

Prezydium Komitetu Głównego, przy udziale jury, rozpatrzyło 6 reklamacji, z których trzy wniosły zmiany do punktacji zawodów I stopnia.

5.5. Zawody II stopnia

Do zawodów II stopnia zakwalifikowano 102 uczniów, którzy osięgnęli w zawodach I stopnia wynik nie mniejszy niż 259 pkt.

Zawody II stopnia odbyły się w dniach 9 – 11 lutego 1995 roku w trzech okręgach (w Toruniu, Wrocławiu i w Warszawie) i uczestniczyli w nich uczniowie z następujacych województw:

w Toruniu – 20 uczniów z województw:

bydgoskiego	– 3,	szczecińskiego	– 4,
gdańskiego	– 8,	toruńskiego	– 3,
olsztyńskiego	– 2,		

we Wrocławiu – 39 uczniów z województw:

bielskiego	– 2,	sieradzkiego	– 1,
częstochowskiego	– 2,	szczecińskiego	– 5,
kaliskiego	– 1,	tarnowskiego	– 4,
katowickiego	– 6,	wałbrzyskiego	– 2,
krakowskiego	– 6,	wrocławskiego	– 3,
opolskiego	– 2,	zielonogórskiego	– 2,
poznańskiego	– 4,		

w Warszawie – 43 uczniów z województw:

białostockiego	– 1,	łomżyńskiego	– 2,
bielskopodlaskiego	– 1,	łódzkiego	– 3,
chełmskiego	– 1,	nowosądeckiego	– 2,
katowickiego	– 1,	radomskiego	– 1,
kieleckiego	– 2,	rzeszowskiego	– 5,
krośnieńskiego	– 1,	siedleckiego	– 1,
lubelskiego	– 4,	st. warszawskiego	– 18.

W zawodach II stopnia najliczniej były reprezentowane szkoły:

1. III L.O. im. Marynarki Wojennej z Gdyni – 6 uczniów,
2. XXVII L.O. im. T. Czackiego z Warszawy – 5,
3. XIII L.O. ze Szczecina – 4,
4. L.O. im. Wł. Jagiełły z Dębicy – 3,
5. V L.O. im. A. Witkowskiego z Krakowa – 3,
6. XXXI L.O. im. L. Zamenhofa z Łodzi – 3,
7. I L.O. im. S. Konarskiego z Mielca – 3,
8. IV L.O. im. T. Kościuszki z Torunia – 3,
9. II L.O. im. St. Batorego z Warszawy – 3.

Najliczniej były reprezentowane szkoły z następujących miast:

Warszawa	– 18 uczniów,	Dębica	– 4 uczniów,
Szczecin	– 9,	Mielec	– 4,
Gdynia	– 6,	Poznań	– 4,
Kraków	– 5,	Toruń	– 4.

Sprawdzanie rozwiązań zadań z zawodów II stopnia było również komputerowo wspomagane – zastosowano łącznie 75 testów do sprawdzenia 510 rozwiązań.

9 lutego odbyła się sesja próbna, na której rozwiązywano nie liczące się do ogólnej klasyfikacji zadanie KLUB PRAWOSKRĘTNYCH KIEROWCÓW.

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

Tabela 3.

Liczba punktów	SZACHOWNICA		MUDSTOCK		TRÓJKĄTY		WIELOKĄTY	
	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo
94 pkt.	1	1,0%	4	3,9%	39	38,2%	3	2,9%
od 93 do 70	14	13,7%	4	3,9%	16	15,7%	5	4,9%
od 69 do 47	29	28,4%	3	2,9%	29	28,4%	10	9,8%
od 46 do 1	45	44,1%	63	61,8%	9	8,8%	61	59,8%
0	13	12,7%	28	27,5%	9	8,8%	23	22,5%

Wyniki za wszystkie 4 zadania są zawarte w Tabeli 4.

Tabela 4.

SUMA	liczba uczniów	procentowo
376 pkt.	0	0,0%
291 pkt – max.	1	1,0%
od 290 do 286 pkt.	0	0,0%
od 285 do 188 pkt.	25	24,5%
od 187 do 1 pkt.	75	73,5%
0 pkt.	1	1,0%

5.6. Zawody III stopnia

Zawody III stopnia odbyły się w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie w dniach od 3 do 7 kwietnia 1995 roku.

W zawodach III stopnia wzięło udział 46 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 161 pkt., z województw:

st.warszawskiego	– 10 uczniów,	krakowskiego	– 1 uczeń,
gdańskiego	– 7,	krośnieńskiego	– 1,
poznańskiego	– 4,	łódzkiego	– 1,
szczecińskiego	– 4,	olsztyńskiego	– 1,
katowickiego	– 3,	opolskiego	– 1,
lubelskie	– 3,	rzeszowskiego	– 1,
tarnowskiego	– 2,	toruńskiego	– 1,
bielskopodlaskie	– 1 uczeń,	wałbrzyskiego	– 1,
bydgoskiego	– 1,	wrocławskiego	– 1,
kieleckiego	– 1,	zielonogórskiego	– 1.

W zawodach III stopnia najliczniej były reprezentowane szkoły:

1. III L.O. im. Marynarki Wojennej z Gdyni – 5 uczniów,
2. L.O. im. Wł. Jagiełły z Dębicy – 2,
3. I L.O. im. K. Marcinkowskiego z Poznania – 2,
4. XIII L.O. ze Szczecina – 2,
5. II L.O. im. St. Batorego z Warszawy – 2,
6. XXVII L.O. im. T. Czackiego z Warszawy – 2,
7. VIII L.O. im. A. Mickiewicza z Poznania – 2,
8. IX L.O. im. K. Hoffmanowej z Warszawy – 2,
9. XIV L.O. im. S. Staszica z Warszawy – 2.

3 kwietnia odbyła się sesja próbna, na której rozwiązywano nie liczące się do ogólnej klasyfikacji zadanie KODOWANIE PERMUTACJI.

Sprawdzanie rozwiązań zadań z zawodów III stopnia było również komputerowo wspomagane – zastosowano łącznie 56 testów do sprawdzenia 230 rozwiązań.

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

Tabela 5.

Liczba punktów	OBCHODZENIE DRZEWA		POCHYLNIA		SKŁADANIE SŁÓW		SZEREGOWANIE ZADAŃ	
	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo	liczba uczniów	procentowo
94	8	17,4%	0	0,0%	11	23,9%	12	26,1%
od 93 do 70	7	15,2%	6	13,0%	12	26,1%	5	10,9%
od 69 do 47	5	10,9%	5	10,9%	14	30,4%	5	10,9%
od 46 do 1	20	43,5%	20	43,5%	8	17,4%	17	37,0%
0	6	13,0%	15	32,6%	1	2,2%	7	15,2%

Wyniki za wszystkie 4 zadania są zawarte w Tabeli 6.

Tabela 6.

SUMA	liczba uczniów	procentowo
380 pkt.	0	0,0%
358 pkt – max.	1	2,2%
od 357 do 286 pkt.	5	10,9%
od 285 do 188 pkt.	18	39,1%
od 187 do 1 pkt.	22	47,8%
0 pkt.	0	0,0%

W dniu 7 kwietnia w sali posiedzeń Rady Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego przy ul. Banacha 2 odbyło się zakończenie II Olimpiady Informatycznej, w trakcie którego ogłoszono wyniki finału Olimpiady i rozdano nagrody. Nagrody ufundowli: Ministerstwo Edukacji Narodowej oraz firmy OPTIMUS, OFEK, Peryt, OEIiZK, IBM Polska i Wydawnictwo Naukowe PWN.

Wyniki zawodów III stopnia II Olimpiady Informatycznej są zawarte w Tabeli 7.

Tabela 7.

L.p.	Nazwisko i imię ucznia szkoła	Nagroda i jej fundator
1	2	3
Laureaci I nagrody		
1	Sawicki Marcin II L.O. im. St. Batorego, Warszawa	Notebook 486DX Color Multimedialny – OPTIMUS
2	Pawlewicz Jakub III L.O. im. Marynarki Wojennej RP, Gdynia	Komputer 486DX Color – OFEK
3	Zieliński Piotr VIII L.O. im. A. Mickiewicza, Poznań	Komputer 486DX Color – MEN
Laureaci II nagrody		
4	Gawron Mikołaj I L.O. im. K. Marcinkowskiego, Poznań	Drukarka atramentowa Lexmark – PERYT
5	Jarnicki Witold V L.O. im. A. Witkowskiego, Kraków	Drukarka atramentowa Lexmark – MEN
6	Mucha Marcin II L.O. im. J. Zamojskiego, Lublin	Drukarka atramentowa HP 520 – OFEK
Laureaci III nagrody		
7	Borowski Adam I L.O. im. M. Curie-Skladowskiej, Starogard Gdański	Borland dBase Compiler – OEIiZK
8	Mędelski-Guz Marcin VI L.O. im. J. i J. Śniadeckich, Bydgoszcz	Borland dBase Compiler – OEIiZK
9	Lipczyński Mateusz III L.O. im. Marynarki Wojennej RP, Gdynia	Borland dBase Compiler – OEIiZK
10	Stefaniak Marcin III L.O. im. Marynarki Wojennej RP, Gdynia	Borland dBase Compiler – OEIiZK
11	Łoś Michał XIII L.O., Szczecin	Borland dBase Compiler – OEIiZK

1	2	3
12	Wawszczak Jarosław VII L.O. im. J. Dąbrowskiego, Zielona Góra	Borland dBase Compiler – OEIiZK
13	Nagórko Andrzej XIV L.O. im. S. Staszica, Warszawa	OS-2 – IBM Polska
14	Brzostek Bartosz L.O. im. Kr. Wł. Jagiełły, Dębica	OS-2 – IBM Polska
15	Krysiński Tomasz I Społeczne L.O., Warszawa	OS-2 – IBM Polska
Wyróżnienia		
16	Fleszar Krzysztof V L.O. im. P. Ściegiennego, Kielce	dBase IV PL – MEN
17	Gordinowicz Przemysław XXXI L.O. im. L. Zamenhoffa, Łódź	Turbo Pascal 7.0 – MEN
18	Kwiatkowski Marek Z.S.Z. nr 1 im. J. Groszkowskiego, Mielec	MS Works 3.0 – MEN
19	Oksiucik Mirosław III L.O. im. Marynarki Wojennej RP, Gdynia	MS Works 3.0 – MEN
20	Stocki Marek XIV L.O. im. Polonii Belgojskiej, Wrocław	MS Windows 3.1 PL – MEN
21	Klin Bartosz XXVII L.O. im. T. Czackiego, Warszawa	MS Windows 3.1 – MEN
22	Mielańczuk Paweł IX L.O. im. K. Hoffmanowej, Warszawa	MS Windows 3.1 – MEN

Pozostali finaliści otrzymali nagrody rzeczowe (dyskietki i książki) ufundowane przez Wydawnictwo Naukowe PWN, oraz przez MEN i OEIiZK.

Jeden z uczniów otrzymał nagrodę *fair play* za skorygowanie pomyłki, jaka pojawiła się w ogłoszonej punktacji zwodów finałowych.

Ogłoszono również komunikat o powołaniu reprezentacji Polski na VII Międzynarodową Olimpiadę Informatyczną do Eindhoven w Holandii i II Olimpiadę Informatyczną Centralnej Europy w Szeged na Węgrzech w składzie:

Marcin Sawicki,

Jakub Pawlewicz,

Piotr Zieliński,

Mikołaj Gawron;

oraz rezerwowi: Witold Jarnicki i Marcin Mucha.

Komitet Główny postanowił wystąpić do Ministerstwa Edukacji Narodowej z wnioskiem o przydzielenie pracowni komputerowej dla III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni, które w finale było reprezentowane przez pięciu uczniów – trzech zostało laureatami a jeden otrzymał wyróżnienie.

Uczniowie wyróżnieni w zawodach II stopnia i zakwalifikowani do zawodów III stopnia Olimpiady mogą być zwolnieni z egzaminu dojrzałości z przedmiotu informatyka na mocy § 9 ust.1 pkt 2 Zarządzenia Ministra Edukacji Narodowej z dnia 14 września 1992r. Sekretariat olimpiady wystawił łącznie 46 zaświadczeń o zakwalifikowaniu do zawodów III stopnia celem przedłożenia dyrekcji szkoły. Laureaci i finaliści mogą być zwolnieni z egzaminów wstępnych do wielu szkół wyższych na mocy uchwał senatów uczelni podjętych na wniosek MEN zgodnie z art. 141 ust. 1 ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65, poz. 385). Sekretariat wystawił łącznie 15 zaświadczeń o uzyskaniu tytułu laureata i 31 zaświadczeń o uzyskaniu tytułu finalisty II Olimpiady Informatycznej celem przedłożenia władzom szkół wyższych.

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

1. mgr Marek Adaszyński
   Ośrodek Doskonalenia Nauczycieli
   w Zielonej Górze
   – nauczyciel laureata Jarosława Wawszczaka.
2. prof. dr hab. Wojciech Guzicki
   Uniwersytet Warszawski
   – nauczyciel laureata Tomasza Krysińskiego.
3. mgr Ryszard Kowal
   II L.O. im. J. Zamojskiego
   w Lublinie
   – nauczyciel laureata Marcina Muchy.
4. mgr inż. Ryszard Kulig
   L.O. im. Króla Wł. Jagiełły
   w Dębicy
   – nauczyciel laureata Bartosza Brzostka oraz finalisty Grzegorza Gawrona.

5. mgr Dzierżysław Malina
   V L.O. im. A. Witkowskiego
   w Krakowie
   – nauczyciel laureata Witolda Jarnickiego.
6. mgr Krzysztof Stefański
   VIII L.O. im. A. Mickiewicza
   w Poznaniu
   – nauczyciel laureata Piotra Zielińskiego oraz finalisty Tomasza Jańczuka.
7. mgr Michał Szuman
   XIII L.O. w Szczecinie
   – nauczyciel laureata Michała Łosia oraz finalisty Marka Rafałowicza.
8. mgr Iwona Waszkiewicz
   VI L.O. im. J. i J. Śniadeckich
   w Bydgoszczy
   – nauczycielka laureata Marcina Mędelskiego-Guz.
9. mgr Jacek Wawrzyniak
   I L.O. im. K. Marcinkowskiego
   w Poznaniu
   – nauczyciel laureata Mikołaja Gawrona oraz finalisty Michała Zawirskiego.
10. dr Piotr Zaremba
    XIV L.O. im. S. Staszica
    w Warszawie
    – nauczyciel laureata Andrzeja Nagórko oraz finalisty Rafała Siejcy.

W dniach 11 i 12 maja br. zorganizowano seminarium dla reprezentacji Polski na zawody międzynarodowe. W zajęciach, które prowadzili dr Krzysztof Diks i mgr Wojciech Plandowski, wzięli udział: Marcin Sawicki, Jakub Pawlewicz, Piotr Zieliński, Mikołaj Gawron, Witold Jarnicki i Marcin Mucha.

6. ZAWODY MIĘDZYNARODOWE

6.1. Olimpiada Informatyczna Centralnej Europy

W Drugiej Olimpiadzie Informatycznej Centralnej Europy CEOI'95 (*Central-European Olympiad in Informatics*) w Szeged na Węgrzech wzięli udział zawodnicy z następujących krajów: Białorusi, Chorwacji, Czech, Estonii, Jugosławii, Litwy, Łotwy, Polski, Rumunii, Słowacji, Słowenii, Ukrainy i Węgier. Mimo zaproszenia nie przybyli zawodnicy z Austrii. W sumie w zawodach wzięło udział 43 uczniów i 1 uczennica z 13 krajów. Zasady doboru uczestników były różne w różnych krajach, choć wszyscy musieli uczęszczać do szkoły w tym roku i mieć nie przekroczone 19 lat. Polską ekipę tworzyli zdobywcy czterech pierwszych miejsc w krajowej II Olimpiadzie, którzy mieli także wziąć udział w Międzynarodowej Olimpiadzie w Eindhoven miesiąc później, w kolejności alfabetycznej: Mikołaj Gawron, Jakub Pawlewicz, Marcin Sawicki i Piotr Zieliński. Opiekunem polskiej ekipy był prof. dr hab. inż. Stanisław Waligórski.

Olimpiada trwała od 29 maja do 3 czerwca, w tym dwa dni, 31 maja i 1 czerwca były przeznaczone na zawody. W każdym z tych dni zawodnicy mieli do rozwiązania po 3 zadania w ciągu 5 godzin. Każde zadanie polegało na napisaniu programu, rozwiązującego pewien problem algorytmiczny, w jednym z następujących języków programowania: Turbo Pascal, Turbo C++ lub QBasic. Polscy zawodnicy, jak większość, posługiwali się językiem Pascal i C++. W sumie za zadania każdego dnia można było otrzymać do 100 punktów, a razem – do 200. Zgodnie z regulaminem tych zawodów, co najwyżej 50% zawodników może zdobyć medale i proporcja złotych medali do srebrnych i brązowych ma być w przybliżeniu jak 1:2:3. O przyznaniu medali poszczególnym zawodnikom, a więc także o ich liczbie, decyduje międzynarodowe jury złożone z kierowników zespołów narodowych, działające pod kierunkiem przewodniczącego, mianowanego przez kraj gospodarza. W tym roku przewodniczył jury dr Arpad Makay.

Ocenianie rozwiązań polegało na sprawdzaniu działania programu zawodnika, w obecności jego i kierownika jego zespołu narodowego, na komputerze za pomocą programu sprawdzającego i zespołu testów – postępowanie znane naszym zawodnikom z zawodów II i III stopnia polskiej olimpiady. Program sprawdzający

działał podobnie jak program stosowany w międzynarodowej olimpiadzie w Sztokholmie rok wcześniej. Ograniczenie czasu działania programu dla każdego testu było jednakowe dla wszystkich testów zadania, jednakowa była też liczba punktów, jaką można było uzyskać za każdy test. Obie liczby były podane w treści zadania. Za każdy test dostawało się albo pełną liczbę punktów, albo zero. Sposób oceniania był więc prostszy niż u nas, ale radykalny: jak się okazało, mając dobrze opracowany algorytm można było dostać 0 punktów za zadanie tylko z powodu pomieszania liter T i Y w nazwie pliku.

Najlepszy wynik, 195 punktów, uzyskał Vladimir Brankov z Jugosławii. Następnymi w kolejności byli: Daniel Kral z Czech – 185 punktów i Martin Hajduch ze Słowacji – 179 punktów. Wszyscy trzej otrzymali złote medale. Następny był Piotr Zieliński (175 punktów), który razem z Jakubem Pawlewiczem(150) i Marcinem Sawickim(146) zdobyli srebrne medale. Mikołaj Gawron uzyskał 126 punktów i zdobył brązowy medal.

Ceremonia rozdania nagród odbyła się w pięknej XIX-wiecznej sali posiedzeń ratusza miasta Szeged. Z portretów na ścianach przyglądali się uroczystości Cesarz Franciszek Józef I i Jego Małżonka, Cesarzowa Elżbieta.

W dniach wolnych od zawodów, 30 maja i 2 czerwca zawodnicy zwiedzali Szeged oraz – normalnie zamknięte dla zwiedzających z powodu konserwacji – wystawę pamiątek historycznych i skansen w Opusztaszer pod Szegedem. Na wystawie pierwszych węgierskich komputerów oraz sprzętu komputerowego produkowanego na Węgrzech do lat siedemdziesiątych spotkali profesora Heinza Zemanka, wybitnego uczonego i pioniera informatyki, konstruktora pierwszego austriackiego komputera.

6.2. VII Międzynarodowa Olimpiada Informatyczna

6.2.1. Przebieg i organizacja

Siódma Międzynarodowa Olimpiada Informatyczna IOI'95 (*International Olympiad in Informatics*), odbyła się w Eindhoven w Holandii w dniach od 26 czerwca do 3 lipca 1995 roku. Polskę reprezentowali laureaci II krajowej Olimpiady Informatycznej: Marcin Sawicki, Jakub Pawlewicz, Piotr Zieliński i Mikołaj Gawron. Opiekunami polskiej ekipy byli: prof. dr hab. inż. Stanisław Waligórski (kierownik ekipy) i mgr Krzysztof Stencel. Tegoroczna zawody miały kilka nowych elementów, zapowiadanych zresztą na długo przedtem*. Najważniejszym z nich było wprowadzenie nowych rodzajów zadań. Zawody odbywały się w ciągu

* Krótka historia międzynarodowych olimpiad informatycznych i dotychczasowe osiągnięcia polskich ekip na tych zawodach zostały opisane w materiałach z pierwszej olimpiady, *I- OI*, str. 30–32.

dwóch dni: 28 i 30 czerwca, w każdym dniu zawodnicy mieli 5 godzin na rozwiązanie 3 zadań. W pierwszym dniu, jedno z zadań należało rozwiązać na papierze, bez pomocy komputera. W drugim – rozwiązaniem jednego z zadań miał być program konwersacyjny, który udziela na ekranie odpowiedzi na dane wpisywane z klawiatury. Rozwiązaniami pozostałych zadań miały być, jak w poprzednich olimpiadach, programy rozwiązujące pewne problemy algorytmiczne i przeznaczone do pracy w trybie wsadowym, to znaczy czytające dane z jednego lub dwu plików wejściowych i zapisujące odpowiedzi w pliku wyjściowym. Aby oswoić zawodników z tym nowym rodzajem zadań, opublikowano zawczasu w biuletynie Olimpiady trzy przykładowe zadania z rozwiązaniami i objaśnieniami. Przytaczamy je w punkcie 9.2.1.

Wśród zadań konkursowych zadanie, które miało być rozwiązywane na papierze, ma znacznie bardziej skomplikowaną treść, niż pozostałe. Nic dziwnego, wymaga ono dość obszernego zapoznania się ze stosowanymi w nim pojęciami i dość szerokich objaśnień, aby można było zrozumieć istotę problemu, bo zadanie wykracza poza normalny zakres pojęć znanych ze szkoły. Ponadto, zadanie składa się z trzech podzadań A, B i C, a w częściach A i C są jeszcze po trzy pytania. Oprócz tego, zadaniu towarzyszy duży komplet formularzy, w których należy wpisywać odpowiedzi.

Pomimo tych różnic, treści zadań konkursowych są podane w takiej kolejności, w jakiej pojawiały się w czasie zawodów: najpierw zadania pierwszego dnia, potem drugiego, a w ramach każdego dnia – najpierw zadania algorytmiczne, a potem zadanie nowego typu.

Jednym z nowych elementów była próba rozpropagowania Olimpiady Informatycznej wśród dziewcząt. Gospodarze ogłosili, że zespoły narodowe, w których będą zarówno dziewczęta jak i chłopcy, będą mogły się składać z pięciu osób zamiast 4, jak tego dotychczas wymagał regulamin. Wywołało to pewne kontrowersje, gdyż procedury kwalifikowania zawodników w wielu krajach nie dawały dostatecznej swobody doboru zawodniczek i zawodników na olimpiadę międzynarodową, albo dlatego, że wybierano tylko zwycięzców krajowych konkursów, albo też obowiązujące regulaminy zakazują wyraźnie wyróżniania osób ze względu na płeć. Jednak wiele krajów skorzystało z tego zaproszenia, tak że w sumie na 51 uczestniczących krajów było 21 zespołów 5 osobowych, 24 zespoły 4 osobowe oraz 5 zespołów mniejszych. Razem w zawodach wzięło udział 203 zawodników, w tym 23 dziewcząt i 180 chłopców.

W ostatecznej klasyfikacji na liście wyników znalazły się 23 zawodniczki i 177 zawodników, razem 200 osób. Trzech zawodników zostało zdyskwalifikowanych za napisanie programów, ingerujących w działanie programu oceniającego.

Poza zawodami gospodarze zaoferowali uczestnikom olimpiady bardzo bogaty program: 27 czerwca, dzień przed zawodami (który jest także dniem aklimatyzacji zawodników przybyłych z prawie wszystkich kontynentów) był przeznaczony na całodzienne zwiedzanie Amsterdamu, połączone z wycieczką statkiem i zakończone wizytą w Rijksmuseum. W dniu pomiędzy zawodami, 29 czerwca, zawodnicy zwiedzali port w Rotterdamie i oglądali mecz baseballu. Po południu wszyscy wzięci udział w przyjęciu wydanym przez władze prowincji Północnego Brabantu, poprzedzonym odczytem, który wygłosił profesor Edsgar Dijkstra. Po zawodach, 1 lipca, wśród wielu atrakcji odbyła się próba ustanowienia rekordu do księgi Guinessa: ustalenie największej liczby osób biorących jednocześnie udział w tej samej grze komputerowej. Wyposażeniem była ściana złożona z kilkuset monitorów widocznych z każdego miejsca trybuny, zajmowanej przez uczestników Olimpiady. Każdy grający siedział na trybunie i był wyposażony w pulpit z joystickiem i przyciskami, za pomocą których mógł wpływać na sytuację na swoim ekranie w tej ścianie. Gra cieszyła się wielkim zainteresowaniem wszystkich i rekord (największa liczba jednocześnie grających) rzeczywiście został ustanowiony.

Wszyscy zawodnicy byli zakwaterowani w domkach na kampingu Kamperveennen około 20 km od Eindhoven, w lesie wśród jezior. Warunki zamieszkania i wypoczynku były doskonałe, było także wiele atrakcji, jak bezpłatny wstęp na pływalnię, plaża, miejsca gier, jednak bardzo napięty program sprawiał, że na korzystanie z tych atrakcji nie było czasu.

6.2.2. Ocenianie prac

Zadanie przeznaczone do rozwiązywania bez komputera było oceniane całkowicie ręcznie, na podstawie tego, co zawodnicy wpisali do formularzy. Pozostałe zadania były oceniane w zwykły już dla zawodów międzynarodowych sposób, to znaczy na komputerze za pomocą specjalnego programu oceniającego i zestawów testów dla każdego zadania. Sprawdzanie rozwiązań zawodnika odbywało się przy nim i w obecności kierownika jego zespołu narodowego, a wyniki sprawdzania można było oglądać na ekranie monitora – w razie potrzeby można było obejrzeć pliki danych testowych, odpowiedzi komputera i postać oczekiwanych rozwiązań. Ograniczenie czasu działania programu wsadowego było w ramach każdego zadania takie samo dla każdego testu. Tak samo jednakowe w ramach zadania były liczby punktów przyznawane za każdy test. Niepomyślna próba przejścia testu kończyła się uzyskaniem 0 punktów.

Protokół oceny rozwiązań zawodnika był podpisywany przez osobę oceniającą i przez kierownika zespołu narodowego. W razie wątpliwości lub niezgodności opinii o wynikach oceniania można było wpisać odpowiednie zastrzeżenia, które

później rozpatrywał Komitet Naukowy, zwykle ponawiając przy tym proces oceniania, już bez zawodnika i jego opiekuna, po czym proponował ocenę. Od tej decyzji Komitetu Naukowego można było się odwołać do Zgromadzenia Ogólnego, składającego się z kierowników wszystkich zespołów narodowych, z przewodniczącym mianowanym przez gospodarzy, czyli przez organizatora olimpiady.

Ocenianie przebiegało na ogół sprawnie i szybko, choć czasami pojawiały się niejasności, np. dla ostatniego zadania drugiego dnia zawodów, kórego rozwiązaniem miał być program konwersacyjny. Spowodowało to konieczność wpisywania odpowiednich uwag do protokołów ocen. Po ponownym sprawdzeniu rozwiązań otrzymaliśmy nowe protokoły i tym razem już oceny prac naszych zawodników zaproponowane przez Komitet Naukowy nie budziły zastrzeżeń. Nasza analiza zadań zgadzała się z tymi ocenami dla wszystkich testów.

6.2.3. Wyniki

Olimpiada zakończyła się niewątpliwym sukcesem zespołu Czech, który zdobył 4 złote medale: Jiri Hajek (168 punktów), Martin Mares (164), Robert Spalek (152) i Pavel Machek (151). Dla Martina Maresa był to już trzeci złoty medal, co stanowi swoisty rekord – poprzednie zdobył na olimpiadach w Sztokholmie w 1994 roku i w Mendozie w 1993 roku. Największą liczbę punktów, 186 i puchar IFIPu zdobył po raz drugi Wiktor Bargaczew z Rosji. Puchar ten został ufundowany, jako najwyższe wyróżnienie, przez Komitet do spraw Kształcenia Międzynarodowej Federacji Przetwarzania Informacji (*International Federation for Information Processing*), światowego zrzeszenia towarzystw informatycznych. Pierwszy raz Wiktor Bargaczew zdobył pierwsze miejsce i ten sam puchar (przechodni) na olimpiadzie w Sztokholmie w 1994 roku, uzyskując wtedy 195 punktów na 200 możliwych. Rok wcześniej, w Mendozie, zdobyło ten puchar 4 najlepszych zawodników, a wśród nich Martin Mares. Wiktor Bargaczew zdobył w Mendozie tylko srebrny medal.

Po 3 złote medale zdobyli zawodnicy z Węgier i Chin, a pozostałe złote medale otrzymali (po jednym) zawodnicy z Bułgarii, Estonii, Japonii, Litwy, Rosji (prócz Bargaczewa), Rumunii, Sri Lanki, Stanów Zjednoczonych i Wielkiej Brytanii.

Polscy zawodnicy uzyskali: 21 miejsce a pierwsze w grupie srebrnych medalistów zdobył Jakub Pawlewicz uzyskując 149 punktów; srebrny medal otrzymał również Piotr Zieliński (za 128 punktów – 44 miejsce), zaś Mikołaj Gawron (113 punktów, 68 miejsce) i Marcin Sawicki (110 punktów, 71 miejsce) zdobyli brązowe medale.

Maksymalne liczby punktów za poszczególne zadania uzyskali:

PAKOWANIE PROSTOKĄTÓW	30 punktów:	Jakub Pawlewicz i Piotr Zieliński

OFERTY SPECJALNE	40 punktów:	Piotr Zieliński
GRA LITEROWA	30 punktów:	Jakub Pawlewicz, Mikołaj Gawron
WYŚCIG ULICZNY	30 punktów:	Marcin Sawicki

W zadaniu PRZEWODY I WŁĄCZNIKI (40 punktów) konwersacja była prowadzona przez program oceniający w sposób najmniej korzystny dla programu zawodnika, tak aby przewlec ją jak najdłużej, wskutek czego w trudniejszych przypadkach liczba kroków przekraczała ustalony limit 900. Trudności sprawiały dwa ostatnie testy, których nie dało się przejść bez dostosowywania się do strategii programu oceniającego. Za najlepszy algorytm dla programu nie konwersacyjnego dla tego zestawu testów można było uzyskać 32 punkty na 40 możliwych i tyle uzyskał Piotr Zieliński.

Za zadanie DRUKOWANIE nikt z polskich zawodników nie uzyskał maksymalnej liczby 30 punktów.

7. TEKSTY I ROZWIĄZANIA ZADAŃ II OLIMPIADY INFORMATYCZNEJ

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

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

Rozwiązania zadań podane przez autorów zadań zawierają najczęściej opis metody, która z jednej strony jest dość prosta, a z drugiej – należy do najlepszych rozwiązań tego zadania. W tych punktach starano się uczynić kompromis między złożonością opisu a jakością rozwiązania. Podana metoda rozwiązania jest zawsze uzasadniona, a czasem zawiera podstawy teoretyczne, na których jest oparta. 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 wejścia/wyjścia. Programy są napisane w języku Turbo Pascal.

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 DRZEWA (Autor: Wojciech Rytter)

Treść zadania DRZEWA

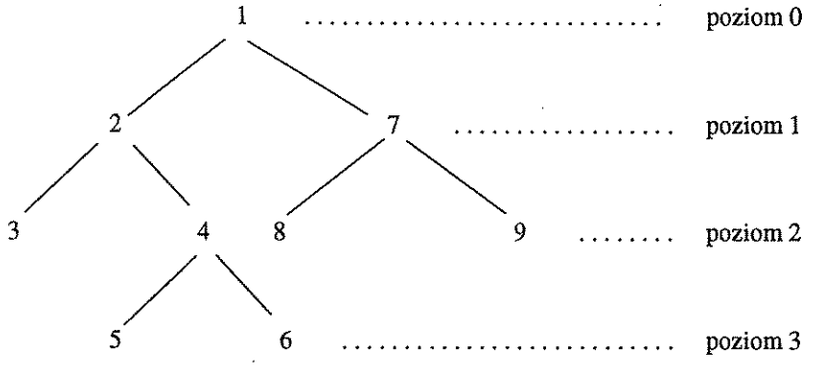
Drzewa występują bardzo często w informatyce. W odróżnieniu od drzew w naturze drzewa w informatyce „rosną do góry nogami"; korzeń jest na górze, a liście są na dole.

Drzewo składa się z elementów zwanych **węzłami**. Jeden z węzłów jest **korzeniem**. Każdy węzeł w (oprócz korzenia) ma swojego (dokładnie jednego) **ojca** v. Jeśli węzeł v jest ojcem w, to w jest synem v. Węzły nie mające synów nazywamy **liśćmi**. Synów węzła w, ich synów, synów ich synów itd. nazywamy **potomkami** węzła w. Każdy węzeł – oprócz korzenia – jest potomkiem korzenia.

Każdy węzeł ma swój **poziom**. Korzeń ma poziom 0, a synowie mają poziom o jeden większy od swoich ojców.

Drzewo jest **pełnym drzewem binarnym** wtedy i tylko wtedy, gdy każdy węzeł ma dokładnie dwóch lub zero synów. W drzewach binarnych rozróżniamy **lewego** i **prawego** syna.

Poniższy rysunek przedstawia przykładowe pełne drzewo binarne. Węzły tego drzewa są ponumerowane w specjalnej kolejności (nazywanej **preorder** w literaturze w języku angielskim). W kolejności tej korzeń ma numer 1, ojciec poprzedza synów, a lewy syn i dowolny jego potomek mają numery mniejsze niż prawy syn i każdy jego potomek.



Jest wiele sposobów zapisu pełnych drzew binarnych z taką numeracją węzłów. Oto trzy z nich.

Zapis genealogiczny

Jest to ciąg liczb. Pierwszy element ciągu jest równy 0 (zero), zaś dla $j > 1$, j-ty wyraz ciągu jest numerem ojca węzła j.

Zapis nawiasowy

Każdemu węzłowi przyporządkowujemy napis złożony z nawiasów. Liściom przyporządkowujemy napis „()". Każdemu z pozostałych węzłów w przyporządkowujemy napis postaci: „(lp)", gdzie l i p oznaczają napisy przyporządkowane odpowiednio lewemu i prawemu synowi w. Napis przyporządkowany korzeniowi jest zapisem nawiasowym drzewa.

Zapis poziomowy

Jest to ciąg poziomów kolejnych liści drzewa (zgodnie z przyjętą numeracją).

Drzewo przedstawione na rysunku można opisać następująco:

Zapis genealogiczny	0 1 2 2 4 4 1 7 7
Zapis nawiasowy	(((()(()))(()()))
Zapis poziomowy	2 3 3 2 2

Zadanie

Napisz program, który wczytuje z pliku tekstowego DRZ.IN ciąg liczb i bada, czy jest on zapisem poziomowym pełnego drzewa binarnego. Jeśli nie jest, to zapisuje w pliku tekstowym DRZ.OUT jeden wyraz NIE, a jeśli jest, to znajduje dwa inne zapisy tego drzewa (genealogiczny i nawiasowy) i zapisuje je w pliku tekstowym DRZ.OUT,

Wejście

W pierwszym wierszu pliku DRZ.IN zapisana jest dodatnia liczba wyrazów ciągu (nie większa niż 2500). W drugim wierszu zapisane są kolejne wyrazy ciągu oddzielone pojedynczymi odstępami.

Liczby w pliku DRZ.IN są zapisane poprawnie. Twój program nie musi tego sprawdzać.

Wyjście

Plik DRZ.OUT powinien zawierać

— tylko jeden wyraz NIE,

albo

— w pierwszym wierszu kolejne wyrazy zapisu genealogicznego oddzielone pojedynczymi odstępami,

— w drugim wierszu zapis nawiasowy, tzn. ciąg nawiasów lewych i prawych bez odstępów między nimi.

Przykłady

Dla pliku DRZ.IN:

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

Twój program powinien utworzyć następujący plik DRZ.OUT:

```
0 1 2 2 1 5 6 6 5
(((()()(((()())())))
```

Dla pliku DRZ.IN:

```
4
1 2 2 3
```

Twój program powinien utworzyć plik DRZ.OUT:

```
NIE
```

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

Rozwiązanie zadania DRZEWA

Zadanie ma charakter kombinatoryczny. Na początku należy się zastanowić, dlaczego kształt drzewa jest jednoznacznie wyznaczony przez jego zapis pozio-

mowy. Z dowodu tego faktu łatwo jest wyprowadzić algorytmy budujące inne reprezentacje drzew. Zajmiemy się więc najpierw tym zadaniem w sensie matematycznym, a następnie omówimy odpowiednie struktury danych i efektywne algorytmy generowania dwóch pozostałych reprezentacji drzew.

W każdym pełnym drzewie binarnym T, przeglądając jego liście od lewej do prawej, w pewnym momencie natrafimy na dwa kolejne liście u i v, które są na tym samym poziomie. Wierzchołki* u, v mają następującą własność:

u, v są braćmi, to znaczy mają wspólnego ojca.

Zatem jeśli usuniemy liście u, v z drzewa, to na ich miejsce wejdzie jako liść ich ojciec z poziomem o jeden mniejszym, niż wynosi ich poziom. W ten sposób otrzymujemy zapis poziomowy drzewa o mniejszej liczbie liści. Postępując tak dalej, albo dochodzimy do sytuacji, gdy pozostaje tylko jeden liść na poziomie 0 – oznacza to, że dane wejściowe były poprawne, albo nie osiągamy takiej sytuacji – czyli ciąg liczb na wejściu nie jest zapisem poziomowym żadnego drzewa.

Skoncentrujemy teraz uwagę na wygenerowaniu zapisu nawiasowego drzewa na podstawie jego zapisu poziomowego x. Odpowiedni algorytm zapiszemy w pseudo-języku programowania.

```
function ZapisNawiasowy(x);
 {x = x_1 x_2 ... x_k dla pewnego k > 0}
 var i, p: Integer;
begin
 if (k = 1) and (x_1 = 0) then return zapis ()
 else begin
  i:= min { j: x_j = x_{j+1} };
  if nie ma takiego i then STOP; {niepoprawne wejście}
  p:=x_i;
  zastąp w x parę x_i x_{i+1} przez x_i - 1;
  y:=ZapisNawiasowy(x);
  wstaw w zapisie y za pozycją p zapis ()();
  return y
 end
end
```

* W treści zadania elementy drzewa są nazywane węzłami, w rozwiązaniu zadania stosujemy natomiast terminologię przyjętą we wszystkich rozwiązaniach, w których występują grafy, i węzły nazywamy wierzchołkami. Podobna terminologia została użyta w materiałach z pierwszej olimpiady *I-OI*.

Algorytm jest bardzo prosty, ale ma pewna wadę – wstawianie w środku napisu (ciągu) jest dość kłopotliwe. Znacznie wygodniej jest zmieniać napis z któregoś jego końców, reprezentując tekst jako listę symboli. Podamy teraz algorytm, który traktuje zapis drzewa jako stos, czyli wykonuje na nim tylko takie operacje, które są związane ze stosem (a więc na jednym końcu zapisu).

Załóżmy, że operacja $x \oplus y$ zastosowana do dwóch tekstów x i y generuje tekst (xy). Tekst ten składa się z połączenia x i y, oraz dopisania na początku nawiasu otwierającego (, a na końcu – nawiasu zamykającego). Przyjmijmy, że zapis poziomowy składa się z k liczb, *Poziom*[i] jest poziomem i-tego elementu w zapisie, oraz na początku mamy *Napis*[i] = (). Dla uproszczenia rozważań załóżmy, że wejściowy zapis poziomowy jest poprawny, czyli odpowiada pełnemu drzewu binarnemu. Ułatwi to znacznie zrozumienie algorytmu. Odpowiednią jego modyfikację, umożliwiającą jednocześnie sprawdzanie poprawności danych wejściowych pozostawiamy Czytelnikowi. Oznaczmy przez PUSH, POP i TOP standardowe operacje na stosie, polegające odpowiednio na: umieszczeniu elementu na stosie, pobraniu elementu ze stosu i odczytaniu elementu znajdującego się na stosie (bez usuwania go). Dodatkowo przyjmijmy, że TOP1 i TOP2 oznaczają dwa najwyższe elementy na stosie.

```
Algorytm Stosowy1;
 Stos:= PustyStos;
 PUSH(1);  Nowy:=k + 1;  Pozycja:=1;
 while (Stos ma co najmniej dwa elementy) or (Pozycja < k) do
  if (Stos ma jeden element) or (Poziom[TOP1] ≠ Poziom[TOP2])
  then begin
   Pozycja:=Pozycja + 1;  PUSH(Pozycja)
  end
  else begin
   Poziom[Nowy]:=Poziom[TOP] - 1;
   Napis[Nowy]:=Napis[TOP2] ⊕ Napis[TOP1];
   POP;  POP;
   PUSH(Nowy);  Nowy:=Nowy + 1
  end;
 Rezultat:=Napis[TOP]
```

Jeśli operacja $\oplus$ jest realizowana za pomocą list (czyli czas połączenia dwóch napisów jest stały, tzn. nie zależy od ich długości), to algorytm ma złożoność liniową względem k, liczby liści w drzewie.

Algorytm *Stosowy*1 można znacznie usprawnić w następujący sposób. Załóżmy, że operacja *Wypisz(Tekst)* wypisuje *Text* na wyjściu (czyli do ustalonego

pliku tekstowego). Po zakończeniu działania algorytmu wypisany tekst (będący złożeniem kolejno wypisywanych fragmentów) jest pełnym poprawnym zapisem nawiasowym. Załóżmy ponownie, że wejściowy zapis poziomowy drzewa jest poprawny, czyli odpowiada pewnemu pełnemu drzewu binarnemu.

```
Algorytm Stosowy2;
 Stos:=PustyStos;
 PUSH(1);
 Wypisz(Poziom[1] lewych nawiasów i jeden prawy nawias);
 Nowy:=k + 1;  Pozycja:=1;
 while (Stos ma co najmniej dwa elementy) or (Pozycja < k) do
  if (Stos ma jeden element) or (Poziom[TOP1] ≠ Poziom[TOP2]) then begin
   Pozycja=Pozycja + 1;
   l:=Poziom[Pozycja] – Poziom[TOP] + 1;
   PUSH(Pozycja);
   Wypisz(l lewych nawiasów i jeden prawy nawias)
  end
  else begin
   Poziom[Nowy]:=Poziom[TOP] – 1;
   POP;  POP;
   Wypisz(jeden prawy nawias);
   PUSH(Nowy);  Nowy:=Nowy + 1
  end;
 Rezultat:=wypisany tekst
```

Zapis genealogiczny drzewa można łatwo wygenerować z zapisu nawiasowego za pomocą następującego algorytmu. Załóżmy, że zapis nawiasowy jest dany w tablicy *Nawias*[*i*] dla $i = 1...m$, gdzie m jest długością zapisu. Zapis genealogiczny jest tworzony w tablicy *Ojciec*.

```
Algorytm Generacja;
 PUSH(1);  Ojciec[1]:=0;  Numer:=1;
 for i:=2 to m do
  if Nawias[i]=prawy nawias then POP
  else begin
   Numer:=Numer + 1;
   Ojciec[Numer]:=TOP;  PUSH(Numer)
  end
```

Algorytmy *Stosowy2* i *Generacja* można połączyć w jeden algorytm, w którym nawiasy w zapisie nawiasowym, generowane przez algorytm *Stosowy2*, są jednocześnie przetwarzane za pomocą algorytmu *Generacja*. Algorytm taki ma jednak dość skomplikowany zapis i jest znacznie mniej strukturalny.

Jeszcze inne rozwiązanie zadania polega na skonstruowaniu najpierw pełnej reprezentacji drzewa, w której dla każdego wierzchołka nie będącego liściem jest określony jego ojciec, lewy i prawy syn. Przyjmijmy, że w tablicach *LewySyn*, *PrawySyn* i *Ojciec* jest pamiętana taka reprezentacja drzewa. Jeśli *LewySyn*[*i*] = 0 i *PrawySyn*[*i*] = 0, to wierzchołek *i* jest liściem w drzewie.

Ważnym elementem tego rozwiązania zadania jest sposób numeracji wierzchołków drzewa. Załóżmy, że wierzchołki drzewa są ponumerowane od 1 do n – numeracja ta na początku nie musi być specjalną numeracją, opisaną np w treści zadania. Dopiero po utworzeniu reprezentacji drzewa możemy przenumerować wierzchołki zgodnie z porządkiem opisanym w treści zadania. Algorytm, który teraz przedstawimy, będzie również działał w sposób *bootom-up* czyli będzie się posuwał od liści w kierunku korzenia. Ponownie zakładamy, że wejściowy zapis poziomowy jest poprawny – bardzo łatwo można zmodyfikować ten algorytm tak, aby sprawdzał również poprawność danych wejściowych.

```
Algorytm Stosowy3;
 Stos:=PustyStos;
 PUSH(1);  Nowy:=k + 1;  Pozycja:=1;
 while (Stos ma co najmniej dwa elementy) or (Pozycja < k) do
  if (Stos ma jeden element) or (Poziom[TOP1] ≠ Poziom[TOP2]) then begin
   Pozycja:=Pozycja + 1;  PUSH(Pozycja)
  end
  else begin
   Poziom[Nowy]:=Poziom[TOP] –1;
   LewySyn[Nowy]:=TOP2;  PrawySyn[Nowy]:=TOP1;
   Ojciec[TOP1]:=Ojciec[TOP2]:=Nowy;
   POP;  POP;
   PUSH(Nowy);  Nowy:=Nowy + 1
  end
```

Dysponując pełną reprezentacją drzewa binarnego można dość prosto wygenerować zapis nawiasowy tego drzewa, przechodząc drzewo metodą DFS – w głąb. Za każdym razem, gdy schodzimy w dół drzewa, generujemy lewy nawias, a gdy się cofamy – generujemy prawy nawias.

Omówienie rozwiązań podanych przez uczniów

Zadanie to nie sprawiło większych trudności uczniom. W rozwiązaniach pojawiły się wszystkie przedstawione powyżej wersje algorytmu, za wyjątkiem algorytmu *Stosowy*1, który ma charakter raczej abstrakcyjno-algebraiczny.

Wielu uczniów zauważyło i uzasadniło, że złożoność całego rozwiązania jest liniowa względem długości zapisu nawiasowego. Zadanie jest dobrą ilustracją wykorzystania stosu i rekurencji, jak również pewnej przewagi własnej realizacji operacji na stosie nad rekurencją.

Testy

Użyte do sprawdzenia rozwiązań testy można podzielić na trzy grupy, w każdej po cztery testy. W pierwszej znalazły się dane o kilku elementach – ich celem było sprawdzenie poprawności rozwiązań. Drugą grupę stanowiły testy o 250 elementach, a trzecia była złożona z danych o maksymalnych rozmiarach i ich zadaniem było testowanie efektywności rozwiązań (a dokładniej, chodziło o wyróżnienie rozwiązań, które mają liniową złożoność). W każdej grupie znajdowały się zarówno dane odpowiadające pełnym drzewom binarnym, jak i dane, które nie były poprawnymi reprezentacjami żadnego drzewa.

7.1.2. Zadanie JEDYNKI I ZERA (Autor*: Andrzej Walat)

Treść zadania JEDYNKI I ZERA

Pewne liczby naturalne mają zapis dziesiętny złożony tylko z jedynek i zer, w którym jest przynajmniej jedna jedynka, na przykład 101. Jeśli liczba naturalna nie ma tej własności, to można próbować pomnożyć ją przez jakąś liczbę naturalną tak, by iloczyn miał tę własność.

Zadanie

Napisz program, który dla każdej liczby naturalnej n nie większej niż 20000, wczytanej z tekstowego pliku danych JED.IN, znajduje jej dodatnią wielokrotność, której zapis dziesiętny składa się z co najwyżej 100 (stu) cyfr, wyłącznie zer lub jedynek i zapisuje tę wielokrotność w pliku tekstowym JED.OUT, albo stwierdza, że takiej wielokrotności nie ma, wpisując w JED.OUT odpowiedź BRAK.

* Zadanie pochodzi ze zbioru zadań rumuńskich. Dr Andrzej Walat opracował treść tego zadania i jest autorem rozwiązań.

Wejście

Plik JED.IN zawiera w pierwszym wierszu całkowitą dodatnią liczbę $K < 1000$, a następnie w kolejnych wierszach ciąg K liczb z zakresu [1..20000], każda w osobnym wierszu.

Liczby w pliku JED.IN są zapisane poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

Każdy kolejny wiersz pliku JED.OUT, począwszy od pierwszego, zawiera

— tylko jeden wyraz BRAK

albo

— dokładnie jedną dodatnią wielokrotność kolejnej danej liczby w postaci ciągu cyfr 0 lub 1 bez odstępów pomiędzy cyframi.

Rozwiązania są zapisane w JED.OUT w takiej samej kolejności jak odpowiednie liczby w JED.IN.

Przykład

Dla pliku JED.IN:

```
6
17
11011
17
999
125
173
```

plik JED.OUT może mieć postać:

```
11101
11011
11101
11111111111111111111111111
1000
1011001101
```

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

Rozwiązanie zadanie JEDYNKI I ZERA

Dla ustalonej liczby naturalnej n, iloczyn $k * n$, gdzie k jest również liczbą naturalną, którego zapis dziesiętny składa się z samych zer i jedynek, będziemy nazywali **zerojedynkową wielokrotnością** liczby n.

Podamy dwa rozwiązania tego zadania, które w pewnym sensie można uznać za dualne: w pierwszym jest tworzona zerojedynkowa wielokrotność liczby n, a w drugim – zerojedynkowa wielokrotność liczby n jest odnajdywana wśród liczb złożonych z zer i jedynek. Oba algorytmy wykonują co najwyżej n operacji i ich komputerowe realizacje są bardzo podobne, dlatego zamieszczamy program tylko dla pierwszego rozwiązania.

W obu rozwiązaniach liczba n jest na początku redukowana. Zauważmy, że jeśli $n = m * 10^w$, gdzie m nie jest podzielne przez 10, to aby znaleźć zerojedynkową wielokrotność n, wystarczy znaleźć zerojedynkową wielokrotność m i dopisać na jej końcu w zer. Podobny fakt jest prawdziwy, jeśli $n = m * 5^w$ (lub $n = m * 2^w$), gdzie m nie jest podzielne przez 5 (odpowiednio, przez 2), gdyż wtedy wystarczy znaleźć zerojedynkową wielokrotność m i pomnożyć ją przez 2^w (odpowiednio, przez 5^w).

Rozwiązanie 1

Algorytm poszukiwania zerojedynkowej wielokrotności liczby n, polegający na mnożeniu n przez kolejne liczby naturalne k i sprawdzaniu, czy iloczyn $k * n$ składa się z samych zer i jedynek, nie jest dobrym rozwiązaniem zadania, ponieważ dla pewnych n z zakresu [1, 20000] najmniejszy mnożnik k taki, że $k * n$ jest zerojedynkową wielokrotnością n, jest większy od 10^{30}.

Jeśli taki algorytm ma dawać wynik w rozsądnym czasie, to należy znaleźć sposób ograniczenia pola poszukiwań. Zauważmy, że jeśli na przykład $n = 17$ oraz $k * n$ jest poszukiwaną wielokrotnością n, to ostatnią cyfrą k musi być 0 lub 3, bo tylko w takich przypadkach ostatnią cyfrą iloczynu będzie 0 lub 1. W istocie można ograniczyć poszukiwania odpowiednich mnożników do liczb k, dla których $k * n$ kończy się cyfrą jeden. Ta obserwacja zmniejsza dziesięciokrotnie pole poszukiwań. Co więcej, można zauważyć, że drugą od końca cyfrą mnożnika k, musi być 5 albo 8, co prowadzi do kolejnego ograniczenia pola poszukiwań, zawężając je do mnożników k z dwucyfrową końcówką 53 lub 83.

Podamy teraz opis algorytmu generowania zerojedynkowej wielokrotności dowolnej liczby n, która nie jest podzielna ani przez 2, ani przez 5.

Algorytm

Budujemy drzewo (poziomami) w następujący sposób.

Krok 1. W wierzchołku początkowym – w korzeniu wpisujemy daną liczbę naturalną n.

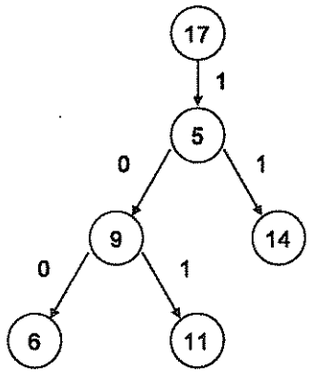
Krok 2. Znajdujemy iloczyn danej liczby n przez liczbę jednocyfrową, który kończy się cyfrą 1. Obcięcie tej liczby, bez ostatniej cyfry, zapisujemy w jedynym wierzchołku drzewa na poziomie 1, a na łuku od korzenia do tego wierzchołka wpisujemy ostatnią cyfrę tej liczby, czyli 1.

Krok 3. Następnie z każdego wierzchołka na poziomie p wyprowadzamy łuki do nowych wierzchołków na poziomie $p + 1$ w następujący sposób:

3.1. Mając daną liczbę w w wierzchołku znajdujemy wszystkie iloczyny liczby n przez liczby jednocyfrowe, które po dodaniu do w tworzą sumę kończącą się cyfrą 0 albo 1. Obcięcia tych sum, bez ostatniej cyfry, zapisujemy w nowych wierzchołkach drzewa, a ostatnie ich cyfry – na łukach prowadzących do tych wierzchołków.

3.2. Rozwijając drzewo poziom za poziomem, do nowych wierzchołków wpisujemy tylko takie liczby, które wcześniej nie wystąpiły.

3.3. Proces konstruowania drzewa kończymy, gdy otrzymamy wierzchołek z liczbą, której zapis składa się z samych zer i jedynek, albo w sytuacji, kiedy nie można już wygenerować żadnego nowego wierzchołka. W pierwszym przypadku, poszukiwaną zerojedynkową wielokrotnością n jest liczba, której zapis składa się z liczby w ostatnim wygenerowanym wierzchołku i wszystkich cyfr na łukach tworzących drogę wstecz od ostatniego wygenerowanego wierzchołka do korzenia drzewa. W drugim przypadku nie ma rozwiązania.



Rysunek 1. Drzewo utworzone dla $n = 17$.

Dla $n = 17$, w jedynym wierzchołku na poziomie 1 zapisujemy liczbę 5, a na łuku od korzenia do tego wierzchołka – cyfrę 1. Następnie z tego wierzchołka można wyprowadzić łuki do dwóch nowych wierzchołków, gdyż $5 + 5 * 17 = 90$ i $5 + 8 * 17 = 141$. W tych wierzchołkach zapisujemy odpowiednio liczby 9 i 14, a na łukach do nich prowadzących – cyfry 0 i 1. Ostatecznie otrzymujemy drzewo pokazane na rys. 1 i na jego podstawie – rozwiązanie 11101.

Własności algorytmu

Własność 1. *Algorytm jest skończony, tzn. budowanie drzewa nie może przedłużać się bez końca.*

Dowód. Liczby w wierzchołkach nie mogą być większe niż dana liczba n, ponieważ:

- każda liczba na pierwszym poziomie drzewa jest wartością wyrażenia $c * n$ div 10, gdzie c jest liczbą jednocyfrową,
- liczby na kolejnych poziomach, wyższych niż jeden, wyznaczamy obliczając wartość wyrażenia $(c * n + w)$ div 10, gdzie w jest liczbą w wierzchołku na niższym poziomie, nie większą niż n.

Ponieważ liczby w wierzchołkach nie mogą się powtarzać, więc dopisywanie nowych wierzchołków drzewa musi zakończyć się po co najwyżej n krokach.

Własność 2. *Jeśli budowanie drzewa kończy się pomyślnie, tzn. wpisaniem do kolejnego wierzchołka liczby, której zapis jest złożony z samych zer i jedynek, to wynik utworzony z tej liczby i cyfr na łukach jest poszukiwaną zerojedynkową wielokrotnością n.*

Dowód. Cyfry zapisywane na kolejnych łukach od korzenia do odpowiedniego wierzchołka końcowego i liczba w tym wierzchołku powstają zgodnie z algorytmem mnożenia pisemnego, tak jak w zwykłym mnożeniu danej liczby n przez liczbę, której cyfry znajdujemy od końca – od najmniej do najbardziej znaczącej.

Własność 3. *Dla każdej liczby $n \leq 20000$ algorytm kończy się pomyślnie znalezieniem jej wielokrotności utworzonej wyłącznie z zer i jedynek.*

Dowód. Empiryczny.

Autor rozwiązania zadania nie umie udowodnić lub obalić następującej hipotezy:

Hipoteza. *Każda liczba naturalna ma zerojedynkową wielokrotność.*

Opisany algorytm znajdowania zerojedynkowej wielokrotności liczby n jest zrealizowany w funkcji `Wynik` w programie `JedynkiZera`.


```
program JedynkiZera;
 const Zakres =20000;
 type TabLog  =array[1..Zakres] of Boolean;
      TabLI   =array[0..9] of LongInt;
      Wsk     =^Element;
      Element =record
                 Ojciec :Wsk;
                 Wierzch:Integer;
                 Strz   :Byte;
                 Nast   :Wsk
               end;
 var Ln,i     :Integer;
     n        :LongInt;
     R        :TabLog;
     Inf,Outf:Text;

 function OK(Liczba:Integer):Boolean;
 begin
  OK:=(Liczba<2) or ((Liczba mod 10 < 2) and
                     (OK(Liczba div 10)))
 end; {OK}

 function Wynik(n:LongInt):String;
  var Koniec                    :Boolean;
      Aktywny,Ostatni,Nowy,Stan:Wsk;
      Wyn,Ogon                  :String;
      Wkn                       :TabLI;
 procedure Redukcja;
  procedure DopiszZera(l:Integer);
  begin
   while n mod l = 0 do begin
    n:=n div l; Ogon:=Ogon+'0'
   end
  end; {DopiszZera}
 begin
  Ogon:='';
  DopiszZera(10);
  DopiszZera(5);
  DopiszZera(2)
```

```
 end; {Redukcja}
 procedure Inicjacja;
  var i:Integer;
 begin
  for i:=1 to Zakres do R[i]:=True;
  New(Aktywny);
  with Aktywny^ do begin
   Ojciec:=Nil;
   Wierzch:=Wkn[1] div 10; Strz:=1;
   Nast:=Nil
  end;
  Ostatni:=Aktywny;
  Koniec:=OK(Aktywny^.Wierzch)
 end; {Inicjacja}

 procedure Przedluz;
  var StWierzch :LongInt;
      NWierzch,i:Integer;
 begin
  StWierzch:=Aktywny^.Wierzch;
  for i:=0 to 1 do begin
   NWierzch:=(StWierzch+Wkn[(10+i-StWierzch mod 10) mod 10])
              div 10;
   if R[NWierzch] then begin
    New(Nowy);
    with Nowy^ do begin
     Ojciec:=Aktywny;
     Wierzch:=NWierzch; Strz:=i;
     Nast:=Nil
    end;
    Ostatni^.Nast:=Nowy; Ostatni:=Nowy;
    R[NWierzch]:=False; Koniec:=OK(NWierzch);
    if Koniec then Break
   end {R[NWierzch]}
  end; {for i=0,1}
  Aktywny:=Aktywny^.Nast
 end; {Przedluz}
```



```
 procedure BudowanieDrzewa;
 begin
  Mark(Stan); Inicjacja;
  while Not Koniec do Przedluz
 end; {BudowanieDrzewa}

 procedure OdczytanieWyn;
 begin
  Str(Ostatni^.Wierzch,Wyn);
  if Ostatni^.Strz=0 then Wyn:=Wyn+'0'
  else Wyn:=Wyn+'1';
  Ostatni:=Ostatni^.Ojciec;
  while Ostatni<>Nil do begin
   if Ostatni^.Strz=0 then Wyn:=Wyn+'0'
   else Wyn:=Wyn+'1';
   Ostatni:=Ostatni^.Ojciec
  end;
  Release(Stan)
 end; {OdczytanieWyn}

 begin {Wynik}
  Redukcja;
  for i:=0 to 9 do Wkn[i*n mod 10]:=i*n;
  if OK(n) then Str(n,Wyn)
  else begin
   BudowanieDrzewa;
   OdczytanieWyn
  end;
  Wynik:=Wyn+Ogon
 end; {Wynik}

begin {JedynkiZera}
 Assign(Inf,'JED.IN'); Reset(Inf);
 Assign(Outf,'JED.OUT'); Rewrite(Outf);
 Readln(Inf,Ln);
 for i:=1 to Ln do begin
  Readln(Inf,n); Writeln(Outf,Wynik(n))
 end;
 Close(Inf); Close(Outf)
end. {JedynkiZera}
```

Opis funkcji `Wynik`

Obliczanie rozpoczynają się od redukcji liczby n. Najpierw odcinamy wszystkie zera na końcu i końcową sekwencję zer zapamiętujemy jako wartość zmiennej `Ogon` typu `String`. Następnie dopisujemy do łańcucha `Ogon` tyle zer, ile razy 5 dzieli n oraz ile razy n dzieli się przez 2. Po tej redukcji n jest liczbą, która nie dzieli się ani przez 2, ani przez 5.

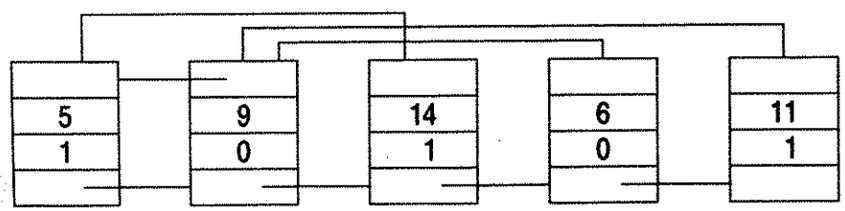
Następnie znajdujemy taką wielokrotność `Wyn` zredukowanego n, której zapis dziesiętny składa się wyłącznie z zer i jedynek – `Wyn` stanowi początkową część wyniku. Po dopisaniu łańcucha `Ogon` do `Wyn` otrzymujemy `Wynik`.

W tym celu najpierw sprawdzamy czy zapis zredukowanego n składa się z samych jedynek i zer. Jeżeli tak, to `Wyn` jest utworzony z cyfr n.

W przeciwnym wypadku budujemy opisane wyżej drzewo wywołując procedurę `BudowanieDrzewa`, a następnie odczytujemy z niego rozwiązanie za pomocą procedury `OdczytanieWyn`. Drzewo reprezentujemy za pomocą typu wskaźnikowego `Wsk`, w którym dodatkowo kolejno dołączane wierzchołki drzewa tworzą listę. Nowe wierzchołki są tworzone poziomami, zaczynając od poziomu pierwszego, zatem drzewo rozrasta się metodą rozszerzania (BFS). Korzeń drzewa pomijamy. Faktycznie, dla dowolnej wartości zredukowanego n może być tylko jeden wierzchołek na poziomie pierwszym, który w konstruowanym drzewie przejmuje rolę korzenia. Każdy wierzchołek drzewa jest rekordem składającym się z czterech pól: w polu `Wierzch` zapisujemy liczbę wpisywaną do wierzchołka drzewa, pole `Strz` zawiera liczbę przypisaną łukowi prowadzącemu do tego wierzchołka, w polu `Ojciec` umieszczamy wskaźnik do ojca, tzn. do wierzchołka, z którego został otrzymany dany wierzchołek, a w polu `Nast` – wskaźnik do wierzchołka następnego na liście, czyli do wierzchołka utworzonego bezpośrednio po tym wierzchołku.

Budując drzewo korzystamy z tablicy `R`, w której odnotowujemy i sprawdzamy występowanie liczb z zakresu od 1 do n w wierzchołkach drzewa.

Reprezentacja drzewa z rys. 1 jest przedstawiona na rys. 2.



Rysunek 2. Reprezentacja drzewa z rys. 1.

Jeśli tworzenie drzewa kończy się sukcesem, czyli w wierzchołku pojawia się liczba zerojedynkowa, to algorytm przechodzi do odczytania wyniku `Wyn`. Pierwszą wartością zmiennej `Wyn` jest pierwsza liczba zerojedynkowa w wierzchołku, do której do końca są dopisywane cyfry 0 lub 1 występujace na łukach drogi od tego wierzchołak do korzenia drzewa.

Analiza wyników programu

Program `JedynkiZera`, dla dowolnej liczby n z zakresu [1..20000] umieszczonej w pliku danych, generuje zerojedynkową wielokrotność tej liczby, której zapis składa się z nie więcej niż 40 cyfr – najdłuższa wygenerowana za pomocą tego programu wielokrotność (dla liczb 10989 i 19998) składa się z 37 cyfr.

Widać stąd, że odpowiedni mnożnik k, taki że $k * n$ jest poszukiwaną wielokrotnością liczby n, w pewnych przypadkach może mieć więcej niż 30 cyfr. Potwierdza to, wyrażone na początku obawy, że poszukiwanie zerojedynkowych wielokrotności pewnych liczb metodą sprawdzania kolejno wszystkich iloczynów danej liczby, może nie dać rozwiązania w rozsądnym czasie. Natomiast podany wyżej algorytm daje wynik po przebadaniu co najwyżej n różnych iloczynów.

Rozwiązanie 2

W tym rozwiązaniu zastosujemy algorytm, którego główna idea polega na generowaniu liczb zerojedynkowych, zaczynając się od liczby 1, i sprawdzaniu, czy któraś z nich nie jest wielokrotnością danej liczby n.

Możemy zacząć od sprawdzenia, czy jednocyfrowa liczba 1 jest wielokrotnością n, a jeśli nie, sprawdzić odpowiednie dwucyfrowe jej przedłużenia 10 i 11, następnie przedłużenia liczb dwucyfrowych o kolejną cyfrę:100, 101, 110, 111 itd. Musimy jednak wziąć pod uwagę, że dla pewnych liczb, ich najkrótsze zerojedynkowe wielokrotności mogą mieć bardzo wiele cyfr, nawet 37, co może oznaczać konieczność wygenerowania i sprawdzenia bardzo dużej ilości liczb zerojedynkowych. Chociaż liczba 2^{37} jest istotnie mniejsza niż 10^{37}, jednak również i w tym przypadku należy znaleźć sposób ograniczenia zakresu poszukiwań, wykluczając pewne liczby.

Zauważmy, że jeśli liczba zerojedynkowa s2 jest przedłużeniem liczby s1 (tzn. cyfry tworzące liczbę s1 występują na poczatku liczby s2) i liczby te dają taką samą resztę z dzielenia przez n (s1 mod n = s2 mod n), to każde przedłużenie liczby s2 daje taką samą resztę z dzielenia przez n, jak odpowiednie przedłużenie (o te same cyfry) liczby s1. Prosty dowód tego faktu pozostawiamy Czytelnikowi.

Przykład

10 mod 7 = 101 mod 7 = 3
100 mod 7 = 1010 mod 7 = 2
1000 mod 7 = 10101 mod 7 = 0.

Z faktu tego wynika, że w procesie generowania kolejnych liczb w poszukiwaniu zerojedynkowej wielokrotności liczby n można pomijać wszystkie liczby dające taką samą resztę z dzielenia przez n jak liczba, która wystąpiła już wcześniej. Oznacza to, że musimy sprawdzić nie więcej niż n liczb.

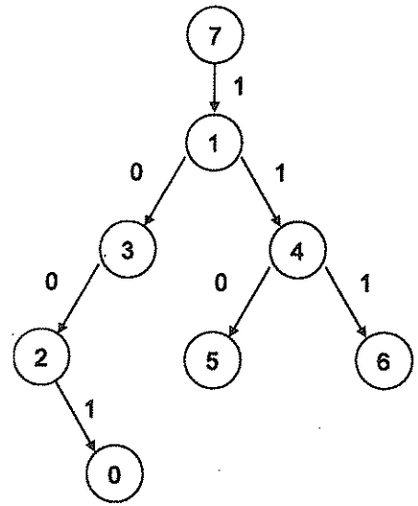
Na przykład poszukując zerojedynkowej wielokrotności liczby 7, generujemy kolejno i sprawdzamy:

1 (1 mod 7 = 1)
10 (10 mod 7 = 3)
11 (11 mod 7 = 4)
100 (100 mod 7 = 2)
110 (110 mod 7 = 5)
111 (111 mod 7 = 6)
1001 (1001 mod 7 = 0) – koniec poszukiwań.

Pominęliśmy liczby: 101, bo 101 mod 7 = 10 mod 7 = 3, a następnie 1000, bo 1000 mod 7 = 111 mod 7 = 6.

Ten sposób generowania zerojedynkowej wielokrotności danej liczby n można przedstawić za pomocą drzewa, takiego jak na rysunku 3. W korzeniu jest dana liczba n = 7. Cyfry 0 i 1 na łukach tworzą zapisy kolejnych liczb, a w każdym wierzchołku od poziomu 1 zapisujemy resztę z dzielenia liczby, utworzonej z cyfr na łukach prowadzących od korzenia do tego wierzchołka, przez liczbę n. Budując drzewo, z wierzchołka, w którym jest reszta r wyprowadzamy łuk z cyfrą 0 do nowego wierzchołka drzewa i zapisujemy w nim nową resztę $10 * r \bmod n$, a w wierzchołku na końcu łuku z cyfrą 1 zapisujemy $(10 * r + 1) \bmod n$. Jednocześnie, w odpowiedniej tablicy odnotowujemy i sprawdzamy każde wystąpienie reszty tak, aby nie powstały dwa wierzchołki drzewa z taką samą resztą.

Algorytm budowania drzew w tym rozwiązaniu (zob. rys. 3) jest bardzo podobny do algorytmu budowania drzew w pierwszym rozwiązaniu. Inne jest tylko znaczenie liczb w wierzchołkach drzewa i inny wzór na ich obliczanie. Również algorytm odczytania zerojedynkowej wielokrotności danej liczby n jest nieco inny niż w rozwiązaniu 1, ponieważ cyfry tej liczby są zapisane kolejno na odpowiednich łukach drzewa od korzenia do odpowiedniego (ostatniego wygenerowanego) liścia. Poza tymi drobnymi, ale istotnymi, różnicami program będący realizacją drugiego algorytmu ma bardzo podobną strukturę i korzysta z takich samych strukturach danych, jak program omówiony w rozwiązaniu 1. Pozostawiamy go więc do samodzielnego wykonania.



Rysunek 3. Drzewo drugiego algorytmu dla n =7.

Omówienie rozwiązań podanych przez uczniów

Wśród około dwudziestu rozwiązań, które uzyskały maksymalną liczbę 100 punktów dominowały różne warianty idei przedstawionej w rozwiązaniu 2, polegającej na obliczaniu dla kolejnych i wszystkich reszt możliwych do uzyskania z dzielenia liczb złożonych z co najwyżej i jedynek i zer, przez daną liczbę n, aż do otrzymania reszty 0.

A oto jeden z przykładów innej niż w rozwiązaniu 2 realizacji tej idei:

„Załóżmy że mamy listę różnych reszt z dzielenia liczb co najwyżej i-cyfrowych przez daną liczbę n. Obliczamy resztę z dzielenia 10^{i+1} przez n i dodajemy ją kolejno (modulo n) do każdej z poprzednio uzyskanych reszt. Jeśli dostaniemy nową resztę, to dopisujemy ją do listy. Jednocześnie zapamiętujemy $i + 1$ w tablicy (np. KIEDY, wyzerowanej na początku obliczeń) pod numerem tej reszty. Ta informacja umożliwia kontrolę, czy kolejna obliczona reszta jest nowa, a na końcu, po otrzymaniu reszty 0, umożliwi odtworzenie cyfr liczby stanowiącej wynik obliczeń (w tablicy KIEDY będziemy odnajdywali pozycje jedynek w zapisie tej liczby)."

Część zawodników usiłowała przyspieszyć działanie swoich programów – czasem skutecznie – wykorzystując stablicowane rozwiązania wybranych przypadków (dla liczb mających szczególnie długie zerojedynkowe wielokrotności, jak np. 9999, 999, 99 i wszystkie wielokrotności tych liczb).

Testy

We wszystkich 20 plikach danych testowych znajdowała się tylko jedna liczba. Były to liczby: 1, 3, 15, 27, 314, 8401, 20000, 1998, 376, 10110, 7992, 11988, 13986, 15984, 9999, 10989, 16983, 18353, 18981.

7.1.3. Zadanie OPTYMALIZACJA DYSKU
(Autorzy: Tomasz Śmigielski i Piotr Krysiuk)

Treść zadania OPTYMALIZACJA DYSKU

Dysk składa się z N **sektorów** ponumerowanych kolejno od 1 do N. Dowolny niepusty ciąg sektorów o kolejnych numerach nazywamy **blokiem**. **Długość bloku** to liczba zawartych w nim sektorów. Bloki są **rozłączne**, jeśli nie mają wspólnego sektora.

Na dysku są zapisane **pliki**. Jeden plik może być zapisany w wielu sektorach, które nie muszą tworzyć jednego bloku. Aby prawidłowo odtworzyć plik trzeba odczytać te sektory we właściwej kolejności. Ten ciąg sektorów wyznacza ciąg rozłącznych bloków, z których każdy zawiera jeden lub więcej sektorów. Sektory wewnątrz każdego bloku czyta się według kolejności numerów.

Opis rozmieszczenia pliku na dysku jest ciągiem par liczb całkowitych dodatnich. Każda taka para składa się z numeru początkowego sektora bloku oraz długości tego bloku.

Przykład

Ciąg par:

7 3
2 1
5 2

informuje, że treść pliku należy odczytać kolejno z sektorów o numerach: 7, 8, 9, następnie: 2, a następnie: 5, 6.

Każdy sektor na dysku może być **wolny** albo jest w nim zapisana część tylko jednego pliku (lub jeden cały plik).

Każdy plik ma unikalny **identyfikator** – dodatnią liczbę całkowitą z zakresu od 1 do P, gdzie P jest liczbą plików na dysku.

Dysk jest **zoptymalizowany**, gdy:

— każdy plik jest pamiętany w jednym bloku (w kolejnych sektorach),
— plik o mniejszym identyfikatorze zajmuje sektory o niższych numerach, niż każdy plik o wyższym identyfikatorze,
— każdy sektor wolny ma numer większy niż każdy sektor zajęty.

Na dysku można wykonywać następujące operacje:

— **kopiowanie** zawartości bloku do rozłącznego z nim bloku o tej samej długości,
— **zamiana** zawartości dwóch rozłącznych bloków o tej samej długości.

Kopiowanie bloku o długości t trwa t mikrosekund. Zamiana zawartości dwóch bloków o długości t trwa $2*t$ mikrosekund.

Polecenie kopiowania pliku ma postać:

```
K początek_bloku nowy_początek_bloku długość_bloku
```

Polecenie zamiany ma postać:

```
Z początek_bloku1 początek_bloku2 długość_bloku
```

gdzie `początek_bloku` oznacza numer pierwszego sektora bloku.

Zadanie

Ułóż program, który:

— wczytuje liczbę sektorów i liczbę plików oraz opisy ich rozmieszczeń na dysku z pliku tekstowego OPT.IN,
— sprawdza, czy dysk jest zoptymalizowany.
Jeśli tak, to zapisuje w pliku OPT.OUT tylko jeden wyraz NIC.
Jeśli nie, to generuje odpowiedni ciąg poleceń powodujących optymalizację dysku w taki sposób, aby czas optymalizacji był jak najkrótszy (nie jest istotna liczba poleceń, tylko łączny czas ich wykonania), po czym zapisuje ten ciąg poleceń w pliku tekstowym OPT.OUT.

Wejście

W pierwszym wierszu pliku danych OPT.IN są zapisane dwie liczby całkowite dodatnie: liczba sektorów na dysku N, nie większa niż 10000, oraz liczba plików P.

Dalej w kolejnych wierszach następują opisy rozmieszczenia plików na dysku.

Każdy opis pliku zawiera w pierwszym wierszu dwie liczby: identyfikator pliku z zakresu od 1 do P oraz dodatnią liczbę rozłącznych bloków, w których zapisany jest ten plik. W następnych wierszach znajduje się opis rozmieszczenia pliku w postaci odpowiedniego ciągu par liczb całkowitych dodatnich, każda para w osobnym wierszu.

Liczby w wierszach są oddzielone pojedynczymi odstępami.

Dane w pliku OPT.IN są zapisane poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

Plik OPT.OUT ma zawierać

— tylko jeden wyraz NIC, gdy dysk jest już zoptymalizowany,

albo

— ciąg poleceń – każde w osobnym wierszu. Każde polecenie musi być zapisane zgodnie z podanym powyżej formatem: najpierw duża litera K albo Z, pojedynczy odstęp, potem trzy liczby całkowite dodatnie oddzielane pojedynczymi odstępami.

Przykład

Dla pliku OPT.IN:

```
200 2
2 2
51 10
41 10
1 2
71 20
11 20
```

plik OPT.OUT może mieć postać:

```
K 21 31 10
K 11 21 10
K 71 1 20
Z 41 51 10
```

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

Rozwiązanie zadania OPTYMALIZACJA DYSKU

Łatwo zauważyć, że zadanie sprowadza się do odpowiedniego przestawiania zawartości sektorów na dysku. (W dalszej treści, zamiast pisać o przenoszeniu zawartości sektorów będziemy czasem pisać w skrócie o przenoszeniu sektorów.) W tym celu skonstruujemy tablicę $Cel[1..N]$ spełniającą dwa warunki:

- jeśli sektor i jest wolny, to $Cel[i] = 0$;
- jeśli sektor i jest zajęty przez pewien plik, to $Cel[i]$ zawiera numer sektora, do którego należy przepisać zawartość sektora i.

Wczytując dane z pliku łatwo obliczyć długość (w sektorach) każdego pliku oraz dla każdego sektora na dysku określić czy jest wolny, a jeśli nie jest, to wyznaczyć numer pliku, do którego należy oraz pozycję tego sektora w pliku (licząc od jeden). Następnie przeglądając jednokrotnie tablicę, w której są zapisane długości plików, możemy dla każdego pliku wyznaczyć łączną długość plików o mniejszych niż ten plik numerach.

Niech sektor i ma pozycję n w pliku p. Na zoptymalizowanym dysku plik p musi być poprzedzony przez pliki o identyfikatorach mniejszych od p. Dlatego docelowa pozycja dla zawartości sektora i na dysku (czyli $Cel[i]$) jest równa sumie n i łącznej długości plików o identyfikatorach mniejszych od p. Aby wyznaczyć wartości w tablicy Cel wystarczy jednokrotne przejrzeć tablicę, w której są zapisane dane o sektorach.

Bardzo cenne jest spostrzeżenie, że kopiowanie bloku o długości d sektorów trwa tyle samo, co d operacji kopiowania bloku o długości 1 (analogicznie jest z zamianą). W związku z tym możemy ograniczyć uwagę do operacji na blokach o długości 1, czyli na pojedynczych sektorach. Znacznie upraszcza to dalsze rozważania, nie powodując pogorszenia jakości rozwiązania. Operacje, które będziemy wykonywać to:

- Skopiowanie zawartości sektora i do wolnego sektora j. Odpowiada to poleceniu K i j 1. Po tej operacji należy wykonać przypisania $Cel[j] := Cel[i]$ oraz $Cel[i] := 0$.
- Zamiana zajętych sektorów i oraz j. Odpowiada to poleceniu Z i j 1. Po tej operacji należy zamienić wartościami $Cel[i]$ oraz $Cel[j]$.

Wykonując te operacje musimy doprowadzić tablicę Cel do stanu, w którym dla każdego i z przedziału od 1 do N zachodzić będzie $Cel[i] = i$ lub $Cel[i] = 0$, tzn. w każdym zajętym sektorze będzie właściwa zawartość.

Będziemy kolejno przeglądać sektory dysku. Dla każdego sektora i, który nie ma właściwej zawartości (tzn. $Cel[i] \neq i$ oraz $Cel[i] \neq 0$), generujemy ciąg poleceń, których wykonanie spowoduje, że zawartość tego sektora (i być może innych również) trafi na swoje miejsce. Wygenerowane polecenia nie będą dotyczyć sektorów, które już mają właściwe zawartości.

Niech i będzie numerem sektora z zawartością nie na właściwym miejscu. Można wyznaczyć ciąg numerów sektorów $a_0, \ldots, a_m$ taki, że $a_0 = i$ oraz dla każdego $k < m$ zachodzi $a_{k+1} = Cel[a_k]$ (czyli zawartość sektora a_0 powinna trafić do sektora a_1, zawartość a_1 do sektora a_2 itd.). Jeśli $Cel[a_m] = 0$, to mamy łańcuch sektorów zakończony sektorem wolnym. W przeciwnym razie powinno być $a_m = a_0$ i sektory w ciągu tworzą cykl.

Jeśli sektory tworzą łańcuch, to wystarczy kolejno kopiować zawartości sektorów na właściwe miejsca. Odpowiada temu następujący ciąg poleceń:

K a_{m-1} a_m 1
K a_{m-2} a_{m-1} 1
...
K a_1 a_2 1
K a_0 a_1 1

Gdy otrzymujemy cykl, to należy rozważyć kilka przypadków. Jeśli $m = 2$, to wystarczy wykonać jedną zamianę, której odpowiada polecenie:

Z a_0 a_1 1

Jeśli cykl jest dłuższy i na dysku nie ma wolnych sektorów, to musimy wykonać $m-1$ zamian:

Z a_{m-1} a_m 1
Z a_{m-2} a_{m-1} 1
...
Z a_1 a_2 1

Najciekawszy jest przypadek cyklu dłuższego niż 2 ($m > 2$), gdy na dysku jest wolny sektor. Niech w będzie numerem wolnego sektora. Możemy skopiować zawartość sektora a_0 do sektora w, a następnie za pomocą $m-1$ kopiowań umieścić zawartości pozostałych sektorów we właściwych miejscach. Teraz wystarczy tylko skopiować zawartość sektor w do a_1 i zawartości wszystkich sektorów cyklu będą na swoich miejscach. Łącznie, wykonamy w ten sposób $m+1$ kopiowań, co będzie trwało $m+1$ mikrosekund. Zauważmy, że wykonanie $m-1$ zamian trwa $2*(m-1)$ mikrosekund. Dla $m = 3$ koszt obu metod jest taki sam, ale dla $m > 3$ kopiowanie jest korzystniejsze. Oto odpowiedni ciąg poleceń:

K a_0 w 1
K a_{m-1} a_m 1
K a_{m-2} a_{m-1} 1
...
K a_1 a_2 1
K w a_1 1

W każdym z omówionych przypadków zawartości wszystkich sektorów wchodzących w skład łańcucha lub cyklu trafiają do właściwych sektorów na dysku i nie istnieje sekwencja poleceń, która umożliwiałaby wykonanie tego w krótszym czasie. Zatem, łączny czas optymalizacji dysku podaną wyżej metodą jest najkrótszy z możliwych.

Fragment programu

Poniżej zamieszczamy fragmenty programu, który jest realizacją podanego algorytmu. Procedura `Optymalizuj` przegląda już wypełnioną tablicę `Cel` i dla każdego sektora, którego zawartość jest w złym miejscu wywołuje procedurę `Przestawiaj`. Procedura `Przestawiaj` znajduje ciąg $a_0, ..., a_m$ i drukuje odpowiedni ciąg poleceń gwarantujących, że zawartości sektorów należących do tego ciągu znajdą się we właściwych sektorach.

```
const Nmax=10000; {maksymalna liczba sektorów}
type Tablica=array[1..Nmax] of Integer;
var N     :Word; {liczba sektorów}
    Wolny:Integer; {numer wolnego sektora, -1 - jeśli brak}
    Cel   :Tablica; {gdzie ma trafić sektor, 0 - dla wolnych}
    Wyj   :Text; {plik wyjściowy}

procedure Przestawiaj(i:Integer);

 procedure Kopiuj(Skad,Dokad:Integer);
  {kopiowanie zawartości jednego sektora ze Skad do Dokad}
 begin
  Writeln(Wyj,'K ',Skad,' ',Dokad,' 1');
  Cel[Dokad]:=Cel[Skad];
  Cel[Skad]:=0;
  Wolny:=Skad
 end; {Kopiuj}

 procedure Zamien(s1,s2:Integer);
  {zamiana sektora s1 z s2}
  var tmp:Integer;
 begin
  Writeln(Wyj,'Z ',s1,' ',s2,' 1');
  tmp:=Cel[s1];
  Cel[s1]:=Cel[s2];
  Cel[s2]:=tmp
 end; {Zamien}

var a  :Tablica; {bufor na przechowywanie ciągu a}
    m  :Integer; {długość wyznaczonego ciągu a}
    n,w:Integer;
```

```
begin {Przestawiaj}
 m:=0;  a[0]:=i;
 repeat
  a[m+1]:=Cel[a[m]]; {a[m+1] - gdzie ma trafić sektor a[m]}
  m:=m+1
 until (Cel[a[m]]=0) or (a[m]=a[0]);
 if Cel[a[m]]=0 then
  for n:=m downto 1 do Kopiuj(a[n-1],a[n])
 else if m=2 then Zamien(a[0],a[1])
      else if Wolny=-1 then
             for n:=m downto 2 do Zamien(a[n],a[n-1]);
           else begin {jest wolny sektor}
            w:=Wolny; {w - numer wolnego sektora}
            Kopiuj(a[0],w);
            for n:=m downto 2 do Kopiuj(a[n-1],a[n]);
            Kopiuj(w,a[1])
           end; {else}
end; {Przestawiaj}

procedure Optymalizuj;
 var Oper:Boolean;
     i   :Integer;
begin
 Oper:=False;
 for i:=1 to N do
  if (Cel[i]<>i) and (Cel[i]<>0) then begin
   Przestawiaj(i);
   Oper:=True  {wykonano operacje na dysku}
  end;
 if not Oper then Writeln(Wyj,'NIC')
end; {Optymalizuj}
```

W kompletnym programie rozwiązujacym zadanie, przed wywołaniem procedury `Optymalizuj` powinny być wykonane następujące czynności:

- wczytanie danych wejściowych,
- wypełnienie tablicy `Cel`,
- zainicjowanie zmiennej `Wolny`,
- otwarcie pliku wyjściowego.

Ich realizację pozostawiamy Czytelnikowi do samodzielnego wykonania.

Złożoność czasowa i pamięciowa algorytmu

Wielkość danych w tym zadaniu można scharakteryzować przez liczbę sektorów na dysku N. Czas potrzebny na analizę danych wejściowych i skonstruowanie tablicy `Cel` jest liniowy ze względu na N. W każdym kroku wystarczy jednokrotnie przeglądnąć zgromadzone dane, których wielkość jest liniowa.

Również czas działania procedury `Optymalizuj` jest liniowy. Wynika to z faktu, że każdy sektor na dysku jest przetwarzany w procedurze `Przestawiaj` co najwyżej raz. Złożoność czasowa podanego algorytmu jest więc liniowa ze względu na liczbę sektorów na dysku.

Złożoność pamięciowa jest także liniowa, ponieważ łączna wielkość wykorzystywanych struktur danych jest proporcjonalna do liczby sektorów na dysku.

Omówienie rozwiązań podanych przez uczniów

Autorzy dużej części rozwiązań zadania zastosowali metodę zachłanną. Najczęściej podany algorytm polegał na wykonywaniu w pętli operacji kopiowania zawartości pojedynczych sektorów do wolnych sektorów oraz zamiany zawartości par sektorów. Przypadek, gdy na nie zoptymalizowanym jeszcze dysku nie można wykonać już więcej takich operacji (co oznacza, że pojawił się cykl) wymaga jednak specjalnego potraktowania. Jeśli na dysku znajduje się wolny sektor, to należy skopiować zawartość jednego z sektorów cyklu na wolne miejsce, dzięki czemu cykl zostaje rozerwany. W przeciwnym przypadku należy zamieniać zawartości sektorów tak, aby przynajmniej zawartość jednego trafiła na właściwe miejsce. Rozwiązania takie miały złożoność nie lepszą niż kwadratowa, względem liczby sektorów.

Pojawiły się również algorytmy bazujące na spostrzeżeniu o łańcuchach i cyklach sektorów wymagających przemieszczenia, podobnie jak w rozwiązaniu wzorcowym. Uczniom z reguły nie sprawiało kłopotu podanie optymalnej kolejności poleceń kopiowania i zamiany zawartości sektorów, wymaganych do zoptymalizowania dysku.

Jednak zaledwie niewielka część nadesłanych rozwiązań była realizacją algorytmu w takiej samej postaci, jak przedstawiliśmy powyżej. Wiele prac korzystało z tablicy `Skad`, przechowującej dla każdego z sektorów miejsce znajdowania się informacji, która po zoptymalizowaniu powinna znaleźć się w tym sektorze. Ponieważ po zoptymalizowaniu dysku zajęty będzie obszar obejmujący `Z` sektorów, istotna jest informacja pamiętana w elementach o indeksach od 1 do `Z` w tej tablicy. Dysponując tablicą `Cel`, inicjalizację takiej struktury danych można wykonać w czasie liniowym względem `N`:

```
for i:=1 to N do
if Cel[i]<>0 then Skad[Cel[i]]:=i;
```

Ciąg (Skad[Skad[...Skad[i]...]],...,Skad[Skad[i]],Skad[i],i) odpowiada ciągowi $a_0, \ldots, a_{m-2}, a_{m-1}, a_m$ z opisu rozwiązania autorów zadania. Dzięki użyciu tablicy Skad można uniknąć tworzenia bufora dla pamiętania sekwencji sektorów wymagających skopiowania lub zamiany zawartości. Możemy skorzystać z faktu, że puste sektory w zajętej części dysku są końcami łańcuchów, a także ze spostrzeżenia, że po optymalizacji łańcuchów pozostaną co najwyżej cykle. Zatem algorytm może mieć następującą postać:

```
 {najpierw optymalizacja lancuchow}
for i:=1 to Z do
 if Cel[i]=0 then {optymalizuj lancuch konczacy się w i};
  {teraz zostaly juz tylko cykle}
for i:=1 to Z do
 if Cel[i]<>i then {optymalizuj cykl zawierajacy i};
```

Idea optymalizacji cykli i łańcuchów nie różni się zasadniczo od przedstawionej w rozwiązaniu wzorcowym. Jedyną różnicą jest inny schemat korzystania z zawartości tablicy Skad w porównaniu z tablicą a. Algorytm ma również złożoność liniową względem liczby sektorów.

Pewna cześć uczniów zrealizowała podobny algorytm, jak przedstawiony powyżej, lecz pominęli konstruowanie tablicy Skad. Powodowało to konieczność liczenia wartości Skad[i] w trakcie optymalizacji, co miało wpływ na pogorszenie złożoności algorytmu do kwadratowej.

Większość uczniów słusznie zauważyła, że wykonywanie operacji na pojedynczych sektorach wyraźnie upraszcza algorytm. Rozwiązania działające na kilkusektorowych fragmentach dysku należały do rzadkości. Częściej w tych rozwiązaniach pojawiały się błędy (np. kopiowanie na koniec dysku zawartości więcej niż jednego sektora podczas rozrywania cyklu, co dawało dłuższy niż optymalny czas optymalizacji).

Mimo, że zgodnie z treścią zadania, długość pliku wynikowego nie miała wpływu na ocenę rozwiązania, część uczniów zrealizowała sklejanie sekwencji kolejnych poleceń odnoszących się do spójnego fragmentu dysku.

Nie wszyscy rozwiązujący zwrócili uwagę na składnię pliku wyjściowego:

- właściwy format poleceń dla sterownika (co najmniej jeden z programów pomijał wypisywanie długości bloku, o czym dość łatwo zapomnieć autorowi programu działającego na pojedynczych sektorach),

- wypisywanie komunikatu NIC, w przypadku zoptymalizowanego dysku (nie wszyscy zawodnicy pamiętali o tym przypadku, a w przynajmniej jednej pracy wypisywano w takiej sytuacji słowo NIE).

Bardzo rygorystycznie traktowano wszelkie odstępstwa od podanej w treści zadania składni pliku wyjściowego i w konsekwencji powyższe błędy niemal zawsze równały się z niezaliczeniem testów.

Testy

Testy można w naturalny sposób podzielić na dwie grupy. Pierwszą tworzyły testy, w których wielkość danych była mała ($N = 10$ lub $N = 20$). Miały one na celu sprawdzenie wszystkich aspektów poprawności rozwiązań. Za rozwiązanie poprawne było uznawane takie, w którym wynikowy ciąg poleceń prowadził do optymalizacji dysku w najkrótszym możliwym czasie. Każdy test badał co najmniej jedną z następujących własności:

- wykrywanie sytuacji, gdy dysk jest zoptymalizowany;
- scalanie plików;
- poprawne działanie, gdy na dysku nie ma wolnych sektorów;
- ustawianie sektorów w plikach w dobrej kolejności;
- scalanie wolnej przestrzeni i umieszczanie jej na końcu dysku;
- ustawianie plików w dobrej kolejności;
- poprawne działanie, gdy nie ma cykli;
- poprawne działanie, gdy są tylko cykle o długości 2;
- poprawne działanie, gdy są tylko cykle o długości większej niż 3.

Drugą grupę stanowiły dwa testy badające efektywność programów. Rozmiary danych wynosiły 5000 i 10000 sektorów. W obu przypadkach wolne sektory stanowiły około 4% wszystkich sektorów. Dysk zawierał zarówno łańcuchy jak i cykle różnej długości.

7.1.4. Zadanie PALINDROMY (Autor: Wojciech Rytter)

Treść zadania PALINDROMY

Słowo jest **palindromem** wtedy i tylko wtedy, gdy nie zmienia się, jeśli czytamy je wspak. Palindrom jest **parzysty**, gdy ma dodatnią parzystą liczbę liter.

Przykładem parzystego palindromu jest słowo abaaba.

Rozkładem słowa na palindromy parzyste jest jego podział na rozłączne części, z których każda jest palindromem parzystym.

Przykład

Rozkładami słowa:

bbaabbaabbbaaaaaaaaaaabbbaa

na palindromy parzyste są:

bbaabb aabbbaaaaaaaaaaaabbbaa

oraz:

bb aa bb aa bb baaaaaaaaaaab bb aa

Pierwszy rozkład zawiera najmniejszą możliwą liczbę palindromów, drugi jest rozkładem na maksymalną możliwą liczbę palindromów parzystych.

Zauważ, że słowo może mieć wiele różnych rozkładów na palindromy parzyste, albo nie mieć żadnego.

Zadanie

Napisz program, który wczytuje słowo z pliku tekstowego PAL.IN i bada, czy da się je rozłożyć na palindromy parzyste. Jeśli nie, to zapisuje w pliku tekstowym PAL.OUT tylko jeden wyraz NIE, a jeśli da się rozłożyć, to zapisuje jego rozkłady na minimalną i maksymalną liczbę palindromów parzystych.

Wejście

Plik wejściowy PAL.IN zawiera jedno słowo złożone z co najmniej 1 i co najwyżej 200 małych liter alfabetu angielskiego, zapisane w jednym wierszu bez odstępów pomiędzy literami.

Zakładamy, że dane słowo jest zapisane w pliku PAL.IN poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

Plik wyjściowy PAL.OUT powinien zawierać:

— tylko jeden wyraz NIE,

albo

— w pierwszym wierszu ciąg słów oddzielonych odstępami stanowiący jeden rozkład danego słowa na minimalną liczbę palindromów parzystych,
— w drugim wierszu jeden rozkład danego słowa na maksymalną liczbę palindromów parzystych.

Przykłady

Dla pliku PAL.IN:

bbaabbaabbbaaaaaaaaaaabbbaa

Twój program powinien utworzyć następujący plik PAL.OUT:

bbaabb aabbbaaaaaaaaaaaabbbaa
bb aa bb aa bb baaaaaaaaaaab bb aa

Dla pliku PAL.IN:

abcde

Twój program powinien utworzyć plik PAL.OUT:

NIE

Dla pliku PAL.IN:

abaaba

Twój program powinien utworzyć plik PAL.OUT:

abaaba
abaaba

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

Rozwiązanie zadania PALINDROMY

Zadanie jest związane z problematyką tekstów i może być sprowadzone do zadania z teorii grafów – po utworzeniu odpowiedniego grafu należy znaleźć w nim najkrótszą lub najdłuższą drogę.

Można zaproponować „naiwne" algorytmy wyznaczania dekompozycji danego słowa x na minimalną i maksymalną liczbę palindromów parzystych; dalej, pierwszy z tych rozkładów nazywamy **minimalnym**, a drugi – **maksymalnym**. Oba algorytmy mają charakter zachłanny – w przypadku maksymalnego rozkładu wybieramy w każdym kroku minimalny palindrom początkowy, a w przypadku minimalnego rozkładu – maksymalny palindrom początkowy.

Algorytm *NaiwnyMax*. Wyznaczanie maksymalnego rozkładu.
while słowo x jest niepuste **do**
  odetnij z x najkrótszy początkowy palindrom

Algorytm *NaiwnyMin*. Wyznaczania minimalnego rozkładu.
while słowo x jest niepuste **do**
  odetnij z x najdłuższy początkowy palindrom

W obu algorytmach, jeśli pozostałe słowo x nie ma na początku palindromu to wypisujemy, że dane na wejściu słowo nie rozkłada się na palindromy.

Najbardziej interesującą własnością zadania jest to, że pierwszy algorytm jest poprawny, podczas gdy drugi nie zawsze generuje minimalne rozkłady słów na palindromy. Ilustruje to następującym przykład.

Przykład

Niech x = aaaaaabbaa. Wtedy algorytm *NaiwnyMax* podzieli to słowo na palindromy: aa aa aa bb aa. Jest to rozkład na maksymalną liczbę palindromów. Natomiast algorytm *NaiwnyMin* podzieli słowo x na palindromy: aaaaaa bb aa, które niestety nie tworzą rozkładu słowa x na minimalną liczbę palindromów, gdyż rozkładem minimalnym jest rozkład na dwa palindromy: aaaa aabbaa.

Algorytm *NaiwnyMax* ma implementację (dość skomplikowaną), która działa w czasie liniowym, a więc rozkład słowa o długości n na maksymalną liczbę palindromów można otrzymać w czasie $O(n)$. Niestety nie znamy algorytmu na minimalny rozkład, działającego w czasie liniowym. Nie przedstawimy tutaj działającej w czasie liniowym implementacji algorytmu znajdowania maksymalnego rozkładu, gdyż i tak złożoność naszego algorytmu dla drugiej części zadania nie jest liniowa. Zamiast tego zbudujemy reprezentację grafową dla tej drugiej części zadania i posłużymy się nią również do znalezienia rozkładu maksymalnego w czasie liniowym.

Niech $x = a_1 a_2 \ldots a_n$. Zbudujmy graf G, którego wierzchołkami są liczby całkowite z przedziału $[0..n]$ i istnieje łuk (połączenie skierowane) od wierzchołka i do wierzchołka j, gdy fragment słowa $x[i,j] = a_{i+1}a_{i+2} \ldots a_j$ jest parzystym palindromem. Za długość łuku (i,j) z i do j przyjmujemy liczbę $j-i$.

Graf G może zawierać dużą liczbę łuków. Dla pary liczb (wierzchołków) i,j można sprawdzić, czy (i,j) jest łukiem w grafie, sprawdzając czy segment $x[i,j]$ w słowie x jest palindromem, za pomocą naiwnego algorytmu działającego w czasie $O(j-i)$. Wtedy całkowita liczba operacji przy tworzeniu grafu G jest rzędu $O(n^3)$, ponieważ jest około n^2 par różnych liczb i,j, które mogą utworzyć łuk.

Aby zwiększyć efektywność tego algorytmu można wstępnie obliczyć wartości w tablicy *Zakres* zdefiniowane następująco:

$$Zakres[i] = \max\{k : x[i-k, i+k] \text{ jest palindromem}\},$$

to znaczy, *Zakres*[i] jest maksymalnym „promieniem" palindromu, którego środek jest zlokalizowany na i-pozycji słowa x. Elementy tablicy *Zakres* można obliczyć w czasie $O(n^2)$, stosując np. algorytm, który liczy zakres palindromu dla każdej pozycji i w słowie osobno. Istnieje algorytm, który wypełnia tę tablicę w czasie liniowym korzystając z prostej kombinatoryki palidromów. Jednakże, z tego samego względu, że koszt całego algorytmu będzie i tak kwadratowy, nie będziemy opisywać tutaj tego dość skomplikowanego algorytmu. Mając wypełnioną tablicę *Zakres* można łatwo zbudować graf G w czasie $O(n^2)$.

Maksymalny rozkład słowa x na palindromy odpowiada najdłuższej (ważonej) drodze w grafie G z wierzchołka 0 do wierzchołka n. Drogę taką znajdujemy stosując algorytm zachłanny: startując z wierzchołka 0 grafu G wybieramy w każdym wierzchołku, w którym się znajdujemy, najkrótszy (w sensie przyporządkowanej mu wagi) łuk aż (ewentualnie) osiągniemy wierzchołek n. Po osiągnięciu wierzchołka n, ciąg wybranych łuków wyznacza rozkład słowa na maksymalną liczbę palindromów.

Minimalny rozkład na palindromy słowa x generujemy znajdując najkrótszą drogę z 0 do n w grafie G. Jeśli nie ma takiej drogi w grafie, to wypisujemy NIE. Istnieje wiele różnych algorytmów znajdowania najkrótszej drogi w grafie. W naszym przypadku graf jest acykliczny i najkrótszą drogę liczymy kolejno od wierzchołków i do n w kolejności odwrotnej do porządku topologicznego tych wierzchołków w grafie G, czyli w kolejności $i = n-1, n-2, \ldots, 0$ (zob. *I-OI*, str. 69-83, gdzie podano więcej szczegółów na temat znajdowania dróg w grafie acyklicznym).

Omówienie rozwiązań podanych przez uczniów

Zadanie to nie sprawiło uczniom większych trudności.

Wielu uczniów zauważyło, że odcinanie najkrótszych palindromów daje maksymalny rozkład. Niektórzy próbowali znaleźć minimalny rozkład „sklejając" najkrótsze palindromy w większe – nie jest to jednak metoda, która zawsze daje poprawne rozwiązanie.

Język teorii grafów (użyty przez nas w opisie rozwiązania) z reguły nie był stosowany jawnie w rozwiązaniach uczniów, chociaż dość często graf był wykorzystywany w sposób pośredni przy rozpatrywaniu punktów podziału słowa na palindromy (punkty takie odpowiadają wierzchołkom opisanego przez nas powyżej grafu G). Dla danej pozycji, liczono minimalną liczbę palindromów w rozkładzie fragmentu słowa, który zaczyna się od tej pozycji. Pozycje były przetwarzane od końca do początku. Odpowiada to metodzie programowania dynamicznego, która tutaj jest po prostu jednym z możliwych sposobów znajdowania najkrótszych dróg w grafie acyklicznym (w przypadku szukania minimalnego rozkładu).

Testy

Rozwiązania były sprawdzane na 12 testach. Tylko w trzech testach liczba liter w słowie była większa niż 100 i rolą tych testów było sprawdzenie efektyw-

ności działania programów uczniowskich. Jeden test ze słowem złożonym z 200 liter miał rozwiązanie 100 100, tzn. najkrótsze i najdłuższe palindromy miały po dwie litery, a drugi był ciągiem 200 liter a.

W pozostałych testach liczba liter w słowie nie była większa niż 34 i ich rolą było sprawdzenie poprawności rozwiązań. Wśród nich były na przykład słowa (w nawiasach podajemy rozwiązanie): a (NIE), aa (1 1), abcdeedcba (1 1), abcdefghij (NIE), abaababb (2 2), alakajak (NIE), aaaabbaaccddcccc (4 8).

7.2. Zawody II stopnia

7.2.1. Zadanie KLUB PRAWOSKRĘTNYCH KIEROWCÓW (Autor: Piotr Chrząstowski-Wachtel)

Treść zadania KLUB PRAWOSKRĘTNYCH KIEROWCÓW

Dana jest prostokątna mapa miasta złożona z $n \times m$ kwadratowych pól ($1 \le n \le 100$ i $1 \le m \le 100$). Wiersze kwadratowych pól tworzących mapę są ponumerowane kolejno od góry do dołu liczbami od 1 do n, a kolumny od lewej do prawej, kolejno od 1 do m. Każde pole jest albo wolne albo zablokowane. Ruch kołowy jest dozwolony tylko po wolnych polach. Z każdego wolnego pola można przejechać na wolne pole przyległe (tzn. takie, które ma z danym polem wspólny bok), nie można jednak zawracać, to znaczy bezpośrednio po przejechaniu z pola X na przyległe pole Y wrócić na X.

Klub Prawoskrętnych Kierowców złożył zamówienie na program komputerowy, który dla dowolnych dwóch różnych pól A oraz B rozstrzyga, czy można dojechać od punktu A do B nie wykonując ani jednego skrętu w lewo, a jeśli tak, znajduje jedną taką drogę o minimalnej długości. Długość drogi jest to liczba wszystkich jej pól. Do drogi od A do B zaliczamy rówmież pola A oraz B.

Zadanie

Ułóż program, który:

- wczytuje z pliku tekstowego KPK.IN dane o prostokątnej sieci pól (tzn. liczbę wierszy n, liczbę kolumn m i stan każdego pola), a następnie współrzędne dwóch różnych pól A i B;
- rozstrzyga, czy istnieje droga bez skrętów w lewo od pola A do B i jeśli nie, to zapisuje w pliku tekstowym KPK.OUT jedno słowo NIE, a jeśli tak, to znajduje jedną taką drogę o minimalnej długości i zapisuje ją w pliku KPK.OUT. Jeśli chociaż jedno z pól A, B nie jest wolne, to odpowiedzią jest NIE.

Wejście

W pierwszym wierszu pliku KPK.IN znajdują się dwie liczby całkowite, oddzielone pojedynczym odstępem: liczba wierszy n oraz liczba kolumn m. W każdym z kolejnych n wierszy pliku znajduje się opis odpowiedniego kolejnego wiersza mapy w postaci jednego słowa o długości m utworzonego z cyfr 0 i 1. Cyfra 0 oznacza, że odpowiednie pole jest wolne a cyfra 1, że jest zablokowane.

Następnie w jednym wierszu są zapisane dwie współrzędne pola A oddzielone pojedynczym odstępem: numer wiersza nie większy niż n oraz numer kolumny nie większy niż m, a w kolejnym wierszu są zapisane, w taki sam sposób, dwie współrzędne pola B.

Dane w pliku KPK.IN są zapisane poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

W pliku KPK.OUT należy zapisać:

- albo jedno słowo NIE, gdy nie istnieje droga bez skrętów w lewo od A do B, lub przynajmniej jedno z pól A, B jest zablokowane,
- albo opis jednej drogi bez skrętów w lewo od A do B o minimalnej długości; w tym przypadku w pierwszym wierszu należy wpisać długość drogi d, a w każdym z kolejnych d wierszy dwie współrzędne kolejnego pola drogi oddzielone pojedynczym odstępem.

Przykłady

Dla następującego pliku KPK.IN:

```
8 9
010011101
011010101
000000000
111010101
101000100
111010101
000000000
101011111
2 6
1 3
```

w pliku KPK.OUT należy zapisać:

```
NIE
```

Dla pliku KPK.IN:

```
8 9
010011101
011010101
000000000
111010101
101000100
111010101
000000000
101011111
2 6
3 8
```

w pliku KPK.OUT należy zapisać:

```
12
2 6
3 6
4 6
5 6
5 5
5 4
4 4
3 4
3 5
3 6
3 7
3 8
```

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

Rozwiązanie zadania KLUB PRAWOSKRĘTNYCH KIEROWCÓW

Jest to zadanie z rodziny zadań, których rozwiązanie polega na znalezieniu drogi w labiryncie, albo – ogólniej – znalezieniu drogi w grafie. W tej klasie zadań mamy zwykle do czynienia z zadanym na płaszczyźnie (lub na szachownicy) zestawem: pól zablokowanych, ścian, pól przejezdnych i dróg, a zadanie polega na znalezieniu drogi łączącej wyróżnione pola i spełniającej zadane kryteria.

Dalej używamy terminologii z teorii grafów, którą posługiwaliśmy się w opisach rozwiązań zadań z I Olimpiady (zob. *I-OI*, str. 72). Dla przypomnienia, **graf skierowany** (lub **digraf**) składa się ze zbioru **wierzchołków** i zbioru **łuków**, będących uporządkowanymi parami wierzchołków.

Zadania o labiryntach rozwiązuje się zwykle przez sprowadzenie ich do zadania o szukaniu drogi w grafie. Należy zatem zastąpić labirynt (mapę) grafem, a następnie wybrać odpowiednią metodę przeszukiwania grafu. W przypadku tego zadania problem został utrudniony przez zdynamizowanie go: istnienie przejścia między dwoma polami zależy nie tylko od statycznej struktury labiryntu, ale od sytuacji dynamicznej – od kierunku najazdu na pole, które mamy opuścić.

Kluczem do znalezienia rozwiązania jest zauważenie, że możliwe jest zdefiniowanie odpowiedniego grafu, którego wierzchołkami są nie tylko pola szachownicy, ale pary (*pole, kierunek najazdu*). Przy takiej interpretacji, dwa wierzchołki grafu (p_1, k_1) oraz (p_2, k_2) są połączone łukiem wtedy i tylko wtedy, gdy z pola p_1 najechawszy na nie z kierunku k_1 można udać się na pole p_2. Wartość k_2 nie jest istotna dla istnienia tego łuku. Każde pole p ma dwie współrzędne, $p = (x, y)$, gdzie w naszym przypadku $1 \le x,\ y \le 100$.

Problemem samym w sobie jest znalezienie odpowiedniej reprezentacji dla takiego grafu w komputerze. Najczęściej stosowanymi reprezentacjami grafu są reprezentacja macierzowa i listowa. W reprezentacji macierzowej tworzymy macierz kwadratową o elementach logicznych (albo ogólnie, rzeczywistych, jeśli chcemy różnicować koszty przejścia po łukach) i tylu wierszach i kolumnach ile jest wierzchołków w grafie. W macierzy tej, zwanej **macierzą sąsiedztwa**, umieszczamy wartość **true** (lub 1), na przecięciu i-tego wiersza i j-tej kolumny wtedy i tylko wtedy, gdy istnieje łuk z i-tego wierzchołka do j-tego, a **false** (lub 0) – w przeciwnym przypadku. Z kolei, w listowej reprezentacji grafu tworzymy tablicę wskaźników do **list sąsiadów**, które zawierają numery wszystkich wierzchołków sąsiednich z danego wierzchołka. Element i-ty w tablicy zawiera wskaźnik do listy sąsiadów i-tego wierzchołka.

Macierz sąsiedztwa jest bardzo kosztowna pod względem zajmowanej pamięci i algorytmy na grafach reprezentowanych w ten sposób są również kosztowne czasowo, szczególnie gdy graf jest grafem rzadkim, czyli takim, w którym liczba łuków jest stosunkowo niewielka, na przykład jest proporcjonalna do liczby wierzchołków. W szczególności graf o ograniczonych i niewielkich stopniach wierzchołków jest rzadki (**stopniem wierzcholka** nazywamy liczbę łuków z nim incydentnych). W takim przypadku, jeśli graf ma N wierzchołków, to wszystkich elementów macierzy sąsiedztwa jest N^2, podczas gdy wartości **true** znajdują się w co najwyżej kN miejscach, dla pewnego stałego czynnika k. Re-

prezentacja listowa grafu może również zawierać pewne dublujące się informacje w przypadku grafów nieskierowanych, gdyż każda krawędź występuje na dwóch różnych listach. Można jednak oszacować, że wielkość pamięci potrzebnej do zapisania grafu w reprezentacji listowej jest proporcjonalna do liczby łuków, czyli dla grafów rzadkich jest znacznie mniejsza, niż wymaga reprezentacja macierzowa.

Czy tworzony przez nas graf jest grafem rzadkim? Widać, że każdy wierzchołek tego grafu ma co najwyżej 2 inne wierzchołki sąsiednie, gdyż wyjazd z pola może nastąpić w co najwyżej jednym z dwóch kierunków (prosto lub w prawo), ustalonych w momencie najechania na pole, z którego wyjeżdżamy. Zatem każda z list sąsiadów tworzonego grafu będzie co najwyżej dwuelementowa.

Powyższe rozważania dotyczą każdego z wierzchołków pośrednich drogi, z wyjątkiem wierzchołka początkowego. Wierzchołek początkowy, odpowiadający polu początkowemu, musimy potraktować osobno, gdyż nie najeżdżamy na to pole, a więc druga składowa w opisie wierzchołka nie jest określona. Z pola początkowego można wyjechać w dowolnym z czterech kierunków, jeśli tylko istnieją (pole początkowe może znajdować się na brzegu). Jeśli chodzi o wierzchołek końcowy, to nie będziemy go już tworzyć – metoda szukania najkrótszej drogi będzie przerywać działanie po osiągnięciu końcowego pola.

Z interpretacji tej wynika, że zadanie sprowadza się do sprawdzenia, czy w tak skonstruowanym grafie istnieje droga wychodząca z wierzchołka początkowego w jednym z czterych możliwych kierunków, a kończąca się w wierzchołku odpowiadającym polu końcowemu. Jeśli istnieje przynajmniej jedna taka droga, to mamy znaleźć najkrótszą z nich, czyli drogę przechodzącą przez najmniejszą liczbę pól.

Znanych jest wiele algorytmów znajdowania najkrótszych dróg w grafie. Większość z nich została opracowana dla grafów z wagami na łukach. Wagi takie ogół są różne dla różnych łuk – jak na przykład odległości między miastami na mapie samochodowej. Takie algorytmy działają również poprawnie w szczególnym przypadku równych wag na łukach, z jakim mamy do czynienia w tym zadaniu. Podamy od razu metodę znajdowania najkrótszych dróg w grafie z równymi wagami na łukach (czyli z wagami równymi 1). Metoda, którą przedstawimy, jest szczególnym przypadkiem algorytmu znajdowania najkrótszych dróg w grafach z wagami na łukach, który podał Dijkstra. O innych algorytmach grafowych, w tym znajdowania szczególnych dróg w grafach, można przeczytać np. w książce W. Lipskiego *Kombinatoryka dla programistów*, WNT, Warszawa 1982. Zobacz również trzeci rozdział w książce: M.M. Sysło, N. Deo, J.S. Kowalik, *Algorytmy optymalizacji dyskretnej w języku Pascal*, Wydawnictwo Naukowe PWN, Warszawa 1994.

Algorytm, który podamy, działa w czasie liniowym ze względu na liczbę łuków (a w naszym przypadku, również ze względu na liczbę wierzchołków) grafu. Algorytm ten jest znany pod nazwą **przeszukiwania grafu wszerz** (po angielsku BFS: **Breadth First Search**).

Przeszukując graf wszerz odwiedzamy kolejno wierzcholki odległe o 1, 2, 3, ... od wierzchołka początkowego. W realizacji tego algorytmu korzysta się ze struktury danych zwanej kolejką. **Kolejka** jest strukturą listową, na której są wykonywane operacje dołączania elementów na koniec, oraz pobierania z początku. Wynika stąd, że elementy są pobierane z kolejki w kolejności ich dołączania. Załóżmy, że operacja `Wstaw(w,q)` dołącza element `w` do kolejki `q`, a operacja `Pobierz(w,q)` – przypisuje zmiennej `w` wartość pierwszego elementu w kolejce `q` i usuwa go z tej kolejki. Będzie nam jeszcze potrzebna funkcja logiczna `Pusta(q)`, sprawdzająca, czy kolejka `q` jest pusta oraz funkcja `Init(q)` – inicjalizująca (pustą) kolejkę.

Algorytm przechodzenia grafu wszerz dla stwierdzenia, czy istnieje w grafie droga z wierzchołka v_1 do wierzchołka v_2 jest bardzo prosty:

```
Init(q);
Znaleziono:=False;
Wstaw(v1,q);
while (not Pusta(q)) and (not Znaleziono) do begin
 Pobierz(w,q);
 if w=v2 then Znaleziono:=True
 else
  while istnieje nieodwiedzony sąsiad wierzchołka w do begin
   niech u będzie nieodwiedzonym sąsiadem w;
   Wstaw(u,q);
   zaznacz u jako odwiedzony
  end
end;
```

Po wykonaniu tego fragmentu algorytmu zmienna `znaleziono` ma wartość `True`, jeśli istnieje droga z v_1 do v_2, a `False` – w przeciwnym razie. Dodatkowo, znaleziona droga będzie drogą najkrótszą. Dla naszych celów ten algorytm musimy nieco rozbudować – musimy bowiem pamiętać wierzchołki tworzonej drogi, gdyż po zakończeniu poszukiwań sukcesem musimy je wypisać, jako część danych wyjściowych.

W tym ostatnim celu użyjemy dodatkowej tablicy `T[1..N] of Integer`, w której pod indeksem `i` będziemy umieszczali numer wierzchołka, który włożył do kolejki wierzchołek o numerze `i`. Innymi słowy, `T[i]` jest numerem wierzchołka, który poprzedza wierzchołek `i` na bieżącej drodze z wierzchołka początkowego do wierzchołka `i`. Odczytując zawartość wypełnionej tablicy po zakończonym sukcesem przeszukiwaniu wszerz odtworzymy ciąg wierzchołków prowadzący najkrótszą drogą z v_1 do v_2, tyle że w odwrotnej kolejności. Wypisanie ich we właściwej kolejności wymaga już niewielkiego wysiłku (zob. procedurę `Droga` w materiałach *I-OI*, str. 123).

Zastanówmy się teraz, jak wygenerować graf odpowiadający możliwym przejściom między polami. Grafu tego nie trzeba tworzyć statycznie, czyli przed szukaniem drogi – można budować go dynamicznie, w trakcie chodzenia po planszy, dodając nowe wierzchołki w miarę osiągania nowych pól. Unikamy w ten sposób np. niepotrzebnego zajmowania się wierzchołkami i łukami, których w danej pozycji na planszy nie możemy wykorzystać.

Przy tworzeniu wierzchołka będziemy nadawali mu nowy numer, większy o 1 od ostatnio nadanego. Ponieważ musimy identyfikować numery wierzchołków z ich współrzędnymi na planszy, trzeba zapamiętać je, powiedzmy w tablicy `Wsp[1..N] of Wspolrzedne`, gdzie `Wspolrzedne = record x,y:Integer; Kier:Strony end;` gdzie $1 \le x, y \le 100$ i `Strony=(N,E,S,W)`.

Aby móc sprawdzić, czy badany wierzchołek o danych współrzędnych został już odwiedzony, deklarujemy dodatkową tablicę

```
Odwiedzone[1..n,1..m] of Kierunki,
```

gdzie

```
Kierunki = set of Strony.
```

Tablicę tę zainicjujemy zbiorami pustymi i ilekroć będziemy chcieli odwiedzić pole o współrzędnych `(i,j)` z kierunku `Kier`, sprawdzimy czy

```
Kier in Odwiedzone[i,j],
```

i jeżeli nie, to wykonamy przypisanie

```
Odwiedzone[i,j]:=Odwiedzone[i,j]+Kier;
```

zaznaczając w ten sposób fakt wejścia na pole `(i,j)` z kierunku `Kier`.

Cały algorytm polega teraz na wyjściu z pola początkowego i przeglądnięciu tworzonego grafu wszerz w poszukiwaniu dojścia do pola końcowego. Jeśli procedura ta kończy działanie sukcesem, to możemy łatwo odtworzyć najkrótszą drogę między podanymi polami.

Omówienie rozwiązań podanych przez uczniów

Zadanie to sprawiło trudność wielu uczestnikom. Na pomysł przeszukiwania grafu wszerz wpadło tylko niewielu. Pozostali najczęściej korzystali z innych

metod szukania najkrótszej drogi w grafie. W szczególności, częściej niż przeszukiwanie wszerz stosowano metodę przeszukiwania grafu w głąb (z nawrotami). Sprowadza się ona do tego, że przeszukiwanie posuwa się tak daleko do przodu, jak tylko jest to możliwe, a w przypadku znalezienia się w wierzchołku, z którego nie ma już wyjścia następuje powrót do ostatnio odwiedzonego wierzchołka i z niego próbujemy dalej iść w głąb.

Przy przeszukiwaniu w głąb można co prawda równie łatwo, jak przy przeszukiwaniu wszerz znaleźć jakąkolwiek drogę między dwoma wierzchołkami, ale gdy szukamy drogi najkrótszej, to w odróżnieniu od przeszukiwania grafu wszerz, pierwsza znaleziona w przeszukiwaniu w głąb droga nie musi być najkrótsza.

Wielu zawodników optymalizowało rekurencyjną realizację przeszukiwania w głąb ucinając rekurencję, gdy tylko droga wydłużyła się powyżej najkrótszej do tej pory znalezionej. Innym przykładem optymalizacji było wstępne przetwarzanie, polegające na rozważeniu szczególnych przypadków ułatwiających w konkretnych przypadkach działanie, ale w ogólności nie podnoszących efektywności algorytmu. Typowymi przykładami było wyliczenie spójności badanych obszarów lub – dla każdego punktu – prostoliniowych odcinków, do których ten punkt należał, i wykorzystywanie w miarę możności tej informacji.

Niektórzy popełniali błędy wynikające z nieuwagi lub błędnego organizowania rekurencji. Na przykład nieuwagą było stwierdzenie, że pole odwiedzone dwukrotnie już nie musi być badane po raz trzeci. Rozwiązaniem jednego z testów była droga, która przechodziła trzykrotnie przez wyróżnione pole (zob. drugi przykład na rys. 1). Przy organizacji rekurencji łatwo było zapomnieć o zaznaczaniu pól już odwiedzonych, co prowadziło do wykładniczych algorytmów, które przeważały wśród rozwiązań.

Pomysł z interpretacją wierzchołków grafu jako pary złożonej z pola i kierunku najazdu wykorzystała większość uczestników. Z jakichś powodów jednak również zdecydowana większość wybierała przeszukiwanie grafu w głąb. Świadczy to być może o popularności myślenia rekurencyjnego, jako naturalnej metodzie rozwiązywania tego typu problemów.

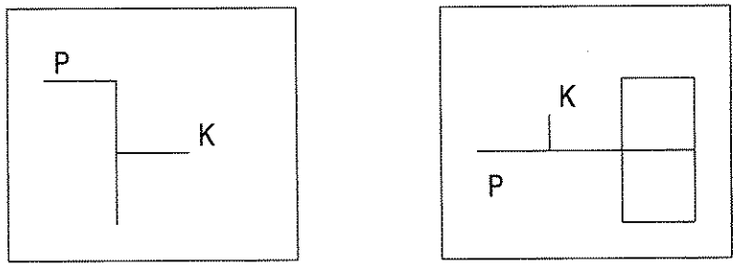
Testy

Testy dla tego zadania miały na celu jak zwykle zbadanie zarówno poprawności, jak i efektywności rozwiązania.

W pierwszym teście pola początkowe i końcowe były przyległe. W drugim (zob. pierwszy przykład na rys. 1) – do pola końcowego można było dojechać wyłącznie wykonując skręt w lewo. Test ten sprawdzał również, czy program nie tworzy drogi zawracającej na jednym polu.

W teście 3 oba pola końcowe drogi leżały na brzegach planszy, a w następnym teście droga ze skrętem w lewo była krótsza od prawidłowego rozwiązania. Test 5 sprawdzał, czy program znajduje najkrótszą drogę z istniejących prawoskrętnych.

Kolejne dwa testy tworzyły sytuacje, w których, aby znaleźć dopuszczalną drogę trzeba było przejechać dwa lub trzy razy przez jedno pole. Ponadto, drugi z nich (zob. drugi przykład na rys. 1) sprawdzał, czy program dopuszcza jazdę po tej samej drodze w przeciwnym kierunku (nie mylić z zawracaniem).



Rysunek 1.

Ostatnią grupę stanowiły testy badające efektywność rozwiązań. Drogi prowadzące do celu były długie. Obszary labiryntu, przez które się przechodziło, były ponadto bogate w wybory, co powodowało że algorytmy z nawrotami miewały kłopoty z wielkością przestrzeni, w której się poruszały. W szczególności jeden z testów wymuszał głębokość rekurencji powyżej 6000, co stanowiło pewną barierę dla niektórych programów. Test 12 stanowiła plansza 100 na 100 pól, na której wszystkie pola były przejezdne.

Ogółem przez komplet testów przeszły poprawnie nieliczne programy, w większości stosujące metodę przechodzenia grafu wszerz.

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

Treść zadania TRÓJKĄTY

W skończonym ciągu dodatnich liczb całkowitych, nie większych niż miliard, reprezentujących długości odcinków chcemy znaleźć trzy takie liczby, że z odpowiadających im odcinków można zbudować trójkąt.

Zadanie

Ułóż program, który sprawdza, czy wśród odcinków – których długości są zapisane w pliku tekstowym TRO.IN – istnieją trzy takie, z których można skonstruować trójkąt i jeśli istnieją, to zapisuje długości tych trzech odcinków w pliku TRO.OUT, a jeśli nie istnieją, to w pliku TRO.OUT zapisuje jedno słowo NIE.

Jeżeli istnieje wiele trójek odcinków o długościach zapisanych w pliku TRO.IN, z których można zbudować trójkąt, Twój program powinien znajdować i wypisywać tylko jedną.

Wejście

W pliku TRO.IN jest zapisany skończony ciąg przynajmniej trzech liczb całkowitych dodatnich, nie większych niż 1000000000, zakończony liczbą 0. Każda liczba jest zapisana w osobnym wierszu. Liczby dodatnie to dane długości odcinków, a liczba 0 stanowi symbol końca danych.

Dane w pliku TRO.IN są zapisane poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

W pliku TRO.OUT powinno być albo jedno słowo NIE, albo trzy długości odcinków z których można zbudować trójkąt, wybrane z pliku TRO.IN, poodzielane pojedynczymi odstępami.

Przykłady

Dla pliku TRO.IN:

```
105
325
55
12555
1700
0
```

w pliku TRO.OUT powinno być jedno słowo:

```
NIE
```

Dla pliku TRO.IN:

```
250
1
105
150
325
```

99999
73
0

przykładem poprawnego rozwiązania jest następująca trójka liczb w pliku TRO.OUT:

250 105 150

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

Rozwiązanie zadania TRÓJKĄTY

Zadanie to na pierwszy rzut oka przypomina zadanie z pierwszego etapu I Olimpiady Informatycznej (zob. materiały *I-OI*, str. 36-43). W tamtym zadaniu chodziło o stwierdzenie, czy dla zadanego ciągu dodatnich liczb wymiernych reprezentujących długości odcinków pewnego zbioru A, z każdych trzech odcinków zbioru A można zbudować trójkąt. W rozwiązaniu wystarczyło znaleźć w zbiorze A dwa najkrótsze odcinki i najdłuższy odcinek oraz sprawdzić, czy można z nich ułożyć trójkąt – zabierało to czas proporcjonalny do liczby elementów zbioru A.

Zasadnicza różnica między tymi dwoma zadaniami jest taka, że tym razem pytamy o **istnienie** w zbiorze A trójki odcinków, z których można zbudować trójkąt. Ponadto, danymi wejściowymi są liczby naturalne, zatem nie pojawia się problem reprezentacji danych.

Rozwiązanie zadania zaczniemy od udowodnienia następującego faktu:

Twierdzenie. *W zbiorze A istnieją 3 odcinki, z których można zbudować trójkąt wtedy i tylko wtedy, gdy w posortowanym niemalejąco ciągu $a_1, a_2, \ldots, a_n$ długości odcinków ze zbioru A istnieje takie k : $3 \le k \le n$, że trzy kolejne liczby a_{k-2}, a_{k-1}, a_k spełniają **nierówność trójkąta**:*

$$a_{k-2} + a_{k-1} > a_k.$$

Dowód. Jeśli w ciągu $a_1, a_2, \ldots, a_n$ istnieją takie trzy kolejne liczby, to z ich posortowania ($a_{k-2} \le a_{k-1} \le a_k$) wynika, że są spełnione również dwie pozostałe nierówności konieczne do zbudowania trójkąta z odpowiadających im odcinków, czyli zachodzą nierówności $a_{k-2} + a_k > a_{k-1}$ oraz $a_{k-1} + a_k > a_{k-2}$.

Załóżmy, że nie istnieją takie trzy kolejne liczby, spełniające nierówność $a_{k-2} + a_{k-1} > a_k$. Wtedy, gdyby istniał trójkąt zbudowany z odcinków o długościach, powiedzmy a_i, a_j, a_k dla pewnych $i < j < k$, to musiałaby być spełniona nierówność $a_i + a_j > a_k$. Ponieważ jednocześnie $j \le k-1$ oraz $i \le k-2$, więc z uporządkowania ciągu $\{a_n\}$ wynikają nierówności $a_k < a_i + a_j \le a_{k-2} + a_{k-1}$, zatem liczby a_{k-2}, a_{k-1}, a_k spełniałyby nierówność trójkąta, czyli sprzeczność.

Z tych rozważań wynika, że zamiast badać wszystkie trójki liczb danych na wejściu (a jest ich $\binom{n}{3} = \frac{n(n-1)n-2)}{6}$, gdy danych jest n), wystarczy dane wejściowe posortować i sprawdzić, czy któraś z kolejnych trójek spełnia nierówność trójkąta. Sortowanie ciągu o długości n dobrą metodą zajmuje czas rzędu $n\log n$, zaś przejrzenie wszystkich kolejnych trójek posortowanych liczb – w najgorszym razie czas proporcjonalny do n.

W naszym zadaniu dane tworzą zbiór liczb naturalnych. Podzbiór ten, z jednej strony jest na tyle duży, że trzeba chwilę pomyśleć, w jaki spoób go posortować (w szczególności nie można zakładać, że wszystkie liczby zmieszczą się w pamięci operacyjnej komputera), z drugiej zaś – okazuje się po przeprowadzeniu analizy, że nie trzeba przetwarzać danych; zadanie to można rozwiązać algorytmem działającym w czasie stałym, czyli niezależnym od liczności zbioru danych wejściowych!

Zauważmy, że zgodnie z treścią zadania, wśród danych może być co najwyżej 10^9 różnych liczb naturalnych. Zastanówmy się, jaki najbardziej liczny zbiór można utworzyć z tych danych tak, aby z żadnych trzech liczb spośród nich nie można było zbudować trójkąta.

Postarajmy się wygenerować ten zbiór metodą zachłanną, dobierając do niego liczby tak małe, jak to jest tylko możliwe. Kolejno wybierane liczby będziemy numerować przez $F_1, F_2, \ldots$ itd. Najmniejszą taką liczbą jest $F_1 = 1$, a następną – $F_2 = 1$. Trzeciej jedynki nie można już jednak dodać, gdyż z trzech jedynek można by utworzyć trójkąt równoboczny. Przyjmujemy zatem $F_3 = 2$. Dwójki nie można już powtórzyć, gdyż 1+2 > 2, zatem przyjmujemy $F_4 = 3$. Teraz kolejną liczbą do zaakceptowania jest 5, gdyż z odcinków o długościach 2, 3 i 3 oraz 2, 3 i 4 można zbudować trójkąt. Widać już metodę: jeśli w zbiorze są liczby $F_1, \ldots, F_k$, ($k \ge 2$), to nie można dodać liczby G mniejszej niż $F_{k-2} + F_{k-1}$, gdyż z odcinków o długościach F_{k-2}, F_{k-1} i G można zbudować trójkąt. Liczbę równą tej sumie można jeszcze dodać i jest to najmniejsza liczba, którą wolno nam dołączyć do naszego zbioru. Zdefiniujmy zatem tworzony ciąg liczb:

$$F = 1,\ F_2 = 1,$$
$$F_k = F_{k-2} + F_{k-1} \text{ dla } k \ge 2. \qquad (1)$$

Kolejnymi wyrazami tego ciągu są liczby:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Zastanówmy się teraz, ile takich liczb można dołączyć do naszego zbioru. Biorąc pod uwagę górne ograniczenie na wielkość danych chcemy obliczyć, jaka jest największa liczba F_n taka, że $F_n < M = 10^9$.

Ciąg, który otrzymaliśmy jest znany pod nazwą **ciągu Fibonacciego**, a liczby F_n – nazywają się **liczbami Fibonacciego**, od nazwiska Leonardo Fibonacciego z Pizy, który w 1202 roku w książce *Liber abaci* podał ich definicję i przeprowadził pierwszą analizę. Od tamtego czasu zadziwiające własności tego ciągu przykuwają uwagę matematyków i informatyków. Ukazuje się nawet specjalne czasopismo *Fibonacci Quarterly* poświęcone wyłącznie liczbom Fibonacciego i ich zastosowaniom.

Pokażemy, że ciąg Fibonacciego rośnie wykładniczo szybko, co oznacza, że niezależnie od wielkości ograniczenia na dopuszczalną wielkość liczby w ciągu, nie można wygenerować zbyt długiego ciągu liczb, z których żadne trzy nie spełniają warunku trójkąta.

W tym celu posłużymy się wzorem na n-tą liczbę Fibonacciego.

Niech φ oraz $\hat{\varphi}$ będą dwoma pierwiastkami równania $x^2 = x + 1$. Przyjmijmy

$$\hat{\varphi} = \frac{1-\sqrt{5}}{2} \quad \text{oraz} \quad \varphi = \frac{1+\sqrt{5}}{2}.$$

Pokażemy indukcyjnie (ze względu na n), że n-ta liczba Fibonacciego jest równa

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \hat{\varphi}^n) \tag{2}$$

Jest to wynik dość zaskakujący: we wzorze pełno liczb niewymiernych, a rezultat działań jest liczbą naturalną.

Baza indukcji musi składać się z dwóch sprawdzeń, gdyż zależność (1) jest zdefiniowana za pomocą dwóch wyrazów o mniejszych indeksach. Aby uprościć obliczenia, wydłużmy ciąg Fibonacciego z lewej strony o $F_0 = 0$. Zauważmy, że nasze równanie rekurencyjne pozostaje prawdziwe, gdyż $F_2 = F_0 + F_1$. Sprawdzamy więc:

$$F_0 = \frac{1}{\sqrt{5}}(\varphi^0 - \hat{\varphi}^0) = 0 \quad \text{oraz} \quad F_1 = \frac{1}{\sqrt{5}}(\varphi^1 - \hat{\varphi}^1) = 1.$$

Założenie indukcyjne: Dla każdego k $(1 \le k < n)$ zachodzi $F_k = \frac{1}{\sqrt{5}}(\varphi^k - \hat{\varphi}^k)$.

Teza indukcyjna: $F_n = \frac{1}{\sqrt{5}}(\varphi^n - \hat{\varphi}^n)$.

Dowód: Ponieważ z definicji $F_n = F_{n-2} + F_{n-1}$, więc na mocy założenia indukcyjnego otrzymujemy:

$$F_n = \frac{1}{\sqrt{5}}(\varphi^{n-2} - \hat{\varphi}^{n-2}) + \frac{1}{\sqrt{5}}(\varphi^{n-1} - \hat{\varphi}^{n-1}) =$$

$$= \frac{1}{\sqrt{5}}(\varphi^{n-2} + \varphi^{n-1} - \hat{\varphi}^{n-2} + \hat{\varphi}^{n-1}) = \frac{1}{\sqrt{5}}(\varphi^{n-2}(1+\varphi) - \hat{\varphi}^{n-2}(1+\hat{\varphi})) =$$

$$= \frac{1}{\sqrt{5}}(\varphi^{n-2}\,\varphi^2 - \hat{\varphi}^{n-2}\,\hat{\varphi}^2) = \frac{1}{\sqrt{5}}(\varphi^n - \hat{\varphi}^n).$$

Jak się dochodzi do takich rezultatów jak (2), to zupełnie inna historia. Metody rozwiązywania zależności rekurencyjnych takich jak (1), czyli otrzymania wzoru (2) na podstawie (1), są opisane na przykład w książce R. Grahama, D.E. Knutha i O. Patashnika *Matematyka konkretna* (Wydawnictwo Naukowe PWN – w przygotowaniu).

Przyjrzyjmy się bliżej wzorowi (2). Wyrażenie $-\hat{\varphi}^2$ bardzo szybko maleje do zera, gdyż $\hat{\varphi} \approx -0.61803$. O wartości F_n decyduje zatem głównie składnik $\frac{1}{\sqrt{5}}\varphi^n$ i z dobrym przybliżeniem można stwierdzić, że n-ta liczba Fibonacciego jest prawie równa tej wartości, czyli $F_n \approx \frac{1}{\sqrt{5}}\varphi^n$ i jest liczbą naturalną, która jest najbliższa wartości $\frac{1}{\sqrt{5}}\varphi^n$.

Aby znaleźć największe n takie, że $F_n < M$, najlepiej zlogarytmować obie strony przybliżonej nierówności $\frac{1}{\sqrt{5}}\varphi^n < M$ przy podstawie φ. Otrzymamy wtedy nierówność

$$n < \log_\varphi M + \log_\varphi \sqrt{5},$$

co dla $M = 1\,000\,000\,000$ daje $n < 44.73\ldots$.

Największą liczbą Fibonacciego nie przekraczającą miliarda jest zatem F_{44}. Wynika stąd, że żaden ciąg, w którym żadne trzy liczby nie są długościami boków trojkąta, nie może zawierać więcej niż 44 elementów. Zatem każdy ciąg liczb z podanego zakresu, którego długość przekracza 44 elementy zawiera 3 liczby będące długościami boków trójkąta.

Upraszcza to znakomicie problemy związane z wielkością danych wejściowych i ułatwia rozwiązanie zadania. Wystarczy bowiem sprawdzić, czy ciąg wejściowy zawiera więcej niż 44 elementy. Jeżeli tak, to na pewno ciąg zawiera 3 liczby będące długościami boków trójkąta i można przerwać wczytywanie danych po wprowadzeniu 45. liczby. Jeżeli danych jest 44 lub mniej, to wystarczy wpisać je do tablicy, posortować w zasadzie dowolnym algorytmem i sprawdzić, czy kolejne trójki liczb w posortowanej tablicy spełniają warunek trójkąta.

Algorytm rozwiązania zadania zatem przedstawia się następująco:

Krok 1. Wczytaj początkowe elementy z danych wejściowych, nie więcej jednak niż 45. Przypisz zmiennej m liczbę wczytanych elementów. Posortuj wczytane liczby w tablicy $T[1..n]$ (praktycznie dowolną metodą – przy tej liczbie elementów czasy sortowania są prawie niezauważalne), gdzie n jest mniejszą z liczb m i 45.

Krok 2. W pętli dla l od 3 do n sprawdź, czy spełniona jest nierówność $T[l-2]+T[l-1]>T[l]$. Jeżeli tak, to zapamiętaj takie l, wyjdź z pętli i wypisz jako dane wyjściowe tę trójkę liczb. Jeżeli takie l nie istnieje, to wypisz NIE.

Ze względu na prostotę podanych czynności nie przytaczamy tu kodu programu.

Omówienie rozwiązań podanych przez uczniów

Zadanie to sprawiło uczniom najmniej kłopotu. Aż 39 ze 102 uczestników drugiego etapu uzyskało maksymalną liczbę punktów, a tylko 7 dostało 0 punktów. Zawodnicy ocenili zadanie jako łatwe.

Sporo uczniów zauważyło związek zadania z liczbami Fibonacciego. Osobnym programem wyliczali n – maksymalną długość wczytywanego ciągu danych, deklarowali w programie odpowiednią tablicę i stosowali którąś z metod sortowania. Niektórzy wyliczyli stałą n źle, zapominając na przykład o możliwości powtórzenia się w ciągu jedynki. Ponieważ jednym z zestawów testowych był maksymalny zbiór liczb, dla którego rozwiązaniem była odpowiedź NIE, więc takie rozwiązania dawały złą odpowiedź dla tego testu.

Innym sposobem ograniczenia ilości danych do sortowania jest zauważenie, że dowolne 3 liczby z przedziału $[2^k, 2^{k+1}-1]$ spełniają warunek trójkąta. Ponieważ kolejnych potęg dwójki mniejszych od 10^9 jest 30, więc sprawdzenie 61 początkowych liczb ciągu daje właściwą odpowiedź. Oszacowanie to jest gorsze, niż bazujące na liczbach Fibonacciego, natomiast różnica między posortowaniem 45 i 61 liczb jest tak niewielka, że w praktyce algorytmy te działały równie szybko.

Wielu uczniów zauważyło, że wystarczy przebadać pewną stałą liczbę długości odcinków. Oszacowania były różne, czasami wynikające z błędnego wyliczenia najmniejszej liczby Fibonacciego przekraczającej miliard, a czasem – z przyjęcia zbyt zgrubnych oszacowań. Zawodnicy wymyślili następujące oszacowania długości maksymalnego ciągu danych, dającego odpowiedź negatywną:

44, 45, 50, 61, 70, 100, 500, 15000, 16000.

Przyjęcie zbyt zgrubnego oszacowania nie powodowało błędu, a jedynie wydłużało działanie programu.

W niektórych rozwiązaniach pożądany efekt osiągnięto przez zastosowanie algorytmu dynamicznego, który po wczytaniu kolejnej liczby umieszczał ją w liście posortowanych już elementów i sprawdzał czy bok o takiej długości domyka trójkąt z sąsiednimi bokami. Można było więc osiągnąć rozwiązanie prawie optymalne nieświadomie. Podobnie, zdarzyły się rozwiązania, w których zawodnicy sortowali dane partiami, np. w tablicach po 15 tysięcy elementów. I znów, nieświadomie otrzymali rozwiązania, które dość szybko kończyły działanie.

W tym zadaniu nie trzeba było przechowywać w pamięci całego ciągu. Kierowanie się schematem „wczytaj dane, przetwórz i wypisz wynik" oznaczało niebezpieczeństwo przepełnienia pamięci w przypadku zbyt długiego pliku.

Testy

Testy, dla których uruchamiano programy były czterech rodzajów.

1. Testy zasadnicze, sprawdzające poprawność algorytmu dla prostych danych.

Testy 1 i 2. Proste, podstawowe zestawy z odpowiedzią TAK, wymagające sortowania; pierwszy z nich był wzięty z treści zadania:

250 1 105 150 325 99999 73 0
234 543 654 12 342 0

Test 3. Prosty zestaw z odpowiedzią NIE: 12 735 34 1902 86 200 0

2. Testy zakresu danych.

Test 4. Zestaw zawierający największą dopuszczalną liczbę, czyli miliard, z odpowiedzią: NIE: 1000000000 98 400000000 50000001 0

Testy 5 i 6. Zestawy zawierające 43 liczby Fibonacciego, kolejno począwszy od F_1 a skończywszy na F_{43} i zakończone: pierwszy z nich – liczbą 565775, która domyka trójkąt na przykład z liczbami F_{28} i F_{29}, a drugi – liczbą F_{44}, która nie domyka żadnego trójkąta i stanowi najdłuższy przykład negatywny. Są to przykłady długich ciągów o różnych odpowiedziach. Poniżej podajemy drugi z nich:

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169
63245986 102334155 165580141 267914296 433494437 701408733 0

3. Test poprawności metody sortowania.

Test 7. Zawierał 16 liczb równych 567. Odpowiedź: pozytywna. Chodziło o wyeliminowanie rozwiązań, które są bezradne wobec równych danych, np. przy sortowaniu lub wstawianiu elementów do posortowanego ciągu.

4. Testy złożoności, wychwytujące algorytmy o dużej złożoności.

Chodziło tu o wychwycenie algorytmów, które stosowały nieefektywne algorytmy, zarówno pod względem czasu obliczeń, jak i zajmowanej pamięci. Testy 8–12 zawierały dane o liczbie elementów wahającej się od 10 000 do 200 000.

7.2.3. Zadanie MUDSTOK BIS (Autor: Andrzej Walat)

Treść zadania MUDSTOK BIS

Władze zrzeszenia Holypolygons z okazji dziesiątej rocznicy pamiętnego pierwszego zlotu swoich członków na błoniach w Mudstock, postanowiły zorganizować wielki festiwal Mudstock bis.

Członkowie zrzeszenia mieszkają w wielu małych osadach Holypolylandii położonych wzdłuż l linii kolejowych (gdzie $1 \le l < 350$) ponumerowanych kolejnymi liczbami naturalnymi od 1 do l. Długość żadnej linii nie przekracza 500 km. Wszystkie linie mają początek w stolicy i biegną promieniście od stolicy ku obrzeżom kraju. Linie te nie krzyżują się. Każda osada nie będąca stolicą leży na dokładnie jednej linii. Na każdej linii liczba osad jest dodatnia i nie większa niż 100. Liczba członków zrzeszenia w jednej miejscowości również nie przekracza 100.

Każdą osadę, która nie jest stolicą identyfikujemy w jednoznaczny sposób za pomocą dwóch współrzędnych (k, n), gdzie k to numer linii, na której leży osada, a n to numer osady na linii. Osady na każdej z linii numerujemy kolejno od stolicy. Przyjmujemy, że stolica (która jest początkiem każdej linii) ma współrzędne (0,0).

Władze zrzeszenia fundują każdemu członkowi bilet kolejowy na powrót z festiwalu do miejsca zamieszkania. Cena biletu jest równa liczbie przejechanych kilometrów. Powstał więc problem, gdzie zorganizować festiwal, aby łączny koszt przejazdu koleją wszystkich członków zrzeszenia z festiwalu do domu był minimalny.

Zadanie

Ułóż program, który:

- wczytuje z pliku MUD.IN dane o sieci kolejowej: liczbę linii kolejowych, liczbę członków zrzeszenia mieszkających w stolicy oraz dla każdej osady liczbę mieszkających w niej członków stowarzyszenia i długość odcinka linii kolejowej od tej osady do najbliższej stacji w kierunku stolicy;
- znajduje miejscowość, w jakiej należy zorganizować festiwal, aby łączny koszt przejazdu koleją wszystkich jego uczestników (członków zrzeszenia) z festiwalu do domu był minimalny,
- oblicza ten minimalny łączny koszt przejazdów;
- zapisuje wyniki w pliku tekstowym MUD.OUT.

Jeżeli dla wielu miejscowości łączny koszt przejazdu koleją wszystkich członków stowarzyszenia z danej miejscowości do domu jest minimalny, to Twój program powinien znajdować jedną z nich.

Wejście

W pierwszym wierszu pliku tekstowego MUD.IN znajdują się dwie liczby całkowite: liczba linii kolejowych l ($1 \le l < 350$) oraz liczba członków zrzeszenia mieszkających w stolicy kraju m ($0 \le m < 100$).

W kolejnych l wierszach znajdują się opisy linii o kolejnych numerach od 1 do l. Każdy opis ma postać ciągu liczb całkowitych pooddzielanych pojedynczymi odstępami.

Najpierw jest podana dodatnia liczba miejscowości, które leżą na danej linii (nie licząc stolicy). Każda kolejna para liczb to: dodatnia odległość kolejnej osady na danej linii od najbliższej osady w kierunku stolicy oraz nieujemna liczba członków zrzeszenia mieszkających w tej osadzie. Bezpośrednio po ostatniej liczbie każdego opisu następuje koniec wiersza.

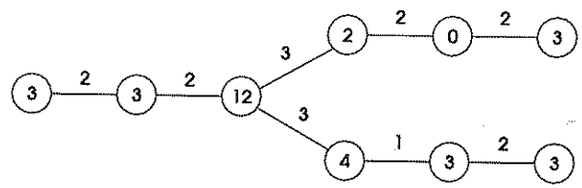
Dane w pliku MUD.IN są zapisane poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

W pierwszym wierszu pliku MUD.OUT należy zapisać minimalny łączny koszt przejazdu koleją wszystkich członków stowarzyszenia z festiwalu do miejsca zamieszkania, a w drugim wierszu współrzędne osady, w której należy zorganizować festiwal.

Przykład

Opisem sieci przedstawionej na rysunku



jest następujcy plik MUD.IN

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

W tym przypadku w pliku MUD.OUT należy zapisać:

```
87
0 0
```

Twój program powinien szukać pliku MUD.IN w katalogu bieżącym i tworzyć plik MUD.OUT również w bieżącym katalogu. Plik zawierający napisany

przez Ciebie program w postaci źródłowej powinien mieć nazwę MUD.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku MUD.EXE

Rozwiązanie zadania MUDSTOCK BIS

Rozwiązanie, które zaproponujemy, jest oparte na następującym lemacie.

Lemat. *Festiwal opłaca się zorganizować poza stolicą tylko wtedy, jeśli na jednej linii mieszka więcej członków stowarzyszenia niż we wszystkich innych osadach kraju.*

Dowód. Jeśli maksymalna liczba członków stowarzyszenia mieszkających na jednej linii *MaxLCL* jest większa niż liczba wszystkich pozostałych członków stowarzyszenia *Reszta* = *LCStow* – *MaxLCL*, to przeniesienie miejsca organizacji festiwalu ze stolicy do pierwszej osady na tej linii oznacza obniżenie kosztów przejazdu każdego z *MaxLCL* członków stowarzyszenia i jednocześnie podrożenie kosztu przejazdu każdego z pozostałych członków o taką samą kwotę równą odległości pierwszej osady na linii od stolicy. Powoduje to obniżenie łącznych kosztów przejazdu o kwotę (*MaxLCL* – *Reszta*)*d*, gdzie *d* jest odległością pierwszej osady na linii od stolicy. W takim przypadku należy zbadać, czy przeniesienie festiwalu do kolejnej osady na linii nie spowoduje dalszej obniżki łącznych kosztów przejazdu, tzn. czy liczba członków stowarzyszenia na odcinku od kolejnej osady na tej linii do jej końca nie jest większa niż liczba wszystkich pozostałych członków stowarzyszenia.

Jeśli *MaxLCL* < *LCStow* – *MaxLCL*, to przeniesienie miejsca festiwalu ze stolicy do osady na linii spowoduje zwiększenie łącznych kosztów przejazdu (dla mniejszej liczby członków stowarzyszenia koszt przejazdu zmaleje, ale dla większej liczby o tyle samo zdrożeje).

Jeśli *MaxLCL* = *LCStow* – *MaxLCL*, to jest wszystko jedno, czy festiwal zorganizujemy w stolicy, czy w pierwszej osadzie na linii, na której mieszka maksymalna liczba członków stowarzyszenia. W takim przypadku może być nawet więcej równie dobrych miejsc organizacji festiwalu, takich że łączny koszt przejazdu jest minimalny.

Algorytm

Krok 1. Wczytujemy dane z pliku tekstowego MUD.IN:

1.1. Dla każdej kolejnej linii obliczamy liczbę członków stowarzyszenia mieszkających na tej linii – *LCS* oraz koszt przejazdu wszystkich członków stowarzyszenia mieszkających na tej linii do stolicy – *KosztLinii*.

1.2. Jednocześnie obliczamy liczbę wszystkich członków stowarzyszenia – – *LCStow* oraz łączny koszt przejazdu wszystkich członków stowarzyszenia do stolicy – *Koszt*. Zapamiętujemy maksymalną liczbę członków stowarzyszenia na jednej linii *MaxLCL*, numer takiej linii – *NrMaxL*, liczbę osad na tej linii *DMaxL*, oraz w tablicy *MaxLinia* dane o wszystkich osadach na tej linii, gdzie $MaxLinia_{i,1}$ jest odległością *i*-tej osady na tej linii od stolicy, a $MaxLinia_{i,2}$ jest liczbą członków stowarzyszenia mieszkających w *i*-tej osadzie.

Krok 2. Następnie, znajdujemy minimalny łączny koszt przejazdów i optymalne miejsce dla zorganizowania festiwalu oraz zapisujemy wyniki do pliku MUD.OUT:

2.1. Jeśli *MaxLCL* ≤ *LCStow* – *MaxLCL*, to zapisujemy wyniki do pliku MUD.OUT: końcową wartość zmiennej *Koszt* oraz współrzędne stolicy – 0 0.

2.2. W przeciwnym przypadku szukamy najlepszego miejsca organizacji festiwalu na linii *NrMaxL*, na której mieszka największa liczba członków stowarzyszenia. W tym celu posuwamy się wzdłuż tej linii w kierunku od stolicy do końca linii dopóty, dopóki liczba członków stowarzyszenia na pozostałym odcinku linii jest większa niż liczba wszystkich pozostałych członków stowarzyszenia. W każdym kroku aktualizujemy odpowiednio wartość zmiennej *Koszt*.

2.3. Na końcu zapisujemy do pliku MUD.OUT wyniki – *Koszt*, *NrMaxL* oraz numer *i* osady, gdzie należy zorganizować festiwal.

Poniżej podajemy fragment programu (procedurę `Wyniki`) będący realizacją drugiego kroku powyższego algorytmu. Opisanie użytych struktur danych oraz pierwszej części algorytmu pozostawiamy do samodzielnego wykonania – nie powinno to nastręczyć większego kłopotu.

```
procedure Wyniki(Koszt,LCStow:LongInt;NrMaxL,MaxLCL:Integer;
                 MaxLinia:OpisLinii);
 var
  f      :Text;
  Reszta:LongInt;
  i      :Integer;
begin
 Assign(f,'MUD.OUT');  Rewrite(f);
 Reszta:=LCStow-MaxLCL;
 i:=0;
 if MaxLCL>Reszta then
  while MaxLCL>Reszta do begin
   Inc(i);
   Koszt:=Koszt+(Reszta-MaxLCL)*MaxLinia[i,1];
```

```
    Reszta:=Reszta+MaxLinia[i,2];
    MaxLCL:=MaxLCL-MaxLinia[i,2]
  end
 else NrMaxL:=0;
 Writeln(f,Koszt);  Write(f,NrMaxL,' ',i);
 Close(f)
end; {Wyniki}
```

Omówienie rozwiązań podanych przez uczniów

Jak wynika z omówionego powyżej rozwiązania, podstawowym warunkiem pełnego sukcesu w tym zadaniu było zauważenie i wykorzystanie następujących faktów:

1. O wyborze najlepszego miejsca organizacji festiwalu rozstrzygają wyłącznie liczby członków stowarzyszenia, mieszkających w poszczególnych osadach (odległości między osadami nie mają z tego punktu widzenia żadnego znaczenia).

2. Nie trzeba pamiętać danych o każdej linii i osadzie – potrzebne są tylko dane o maksymalnej linii.

3. Poszukiwany minimalny koszt przejazdów i niektóre wyniki częściowe mogły swoją wielkością przekroczyć zakres prostego typu `Integer`, należało więc użyć typu `LongInt`.

Pominięcie, któregokolwiek z tych trzech faktów powodowało, że program rozwiązujący zadanie stawał się nieefektywny, zajmował zbyt dużo pamięci lub dla pewnych danych obliczał niepoprawny minimalny koszt przejazdów. Niestety większość uczestników drugiego etapu zawodów popełniła któryś z tych błędów.

Testy

Rozwiązania uczniów były oceniane na podstawie wyników działania na 14 testach. W sześciu pierwszych – rozmiary danych były małe i liczba linii była mniejsza niż 100. Pierwszy test zawierał dane z przykładu w treści zadania. Celem tych testów było sprawdzenie poprawności algorytmów. Nawet nieefektywne ale poprawnie działające programy powinny były dać prawidłowy wynik w dość krótkim czasie. Testy 7 – 12 zawierały dane o coraz większych rozmiarach: 150 linii w testach 7 i 8, 200 linii w testach 9 i 10, 300 linii i po 99 lub 100 osad na każdej linii w testach 11 i 12. Zadaniem tych testów było określenie ocen na podstawie dokładnego rozróżnienia złożoności czasowej i pamięciowej programów. W plikach 13 i 14 znajdowały sie dane o maksymalnym dopuszczalnym rozmiarze (349 linii, po 99 lub 100 osad na każdej linii).

7.2.4. Zadanie POKRYWANIE SZACHOWNICY (Autor: Wojciech Rytter)

W dalszej treści używamy skróconej nazwy tego zadania SZACHOWNICA.

Treść zadania SZACHOWNICA

Dana jest szachownica o rozmiarach n na n, gdzie n jest nieparzystą liczbą całkowitą spełniającą nierówność $3 \le n < 50$, oraz zestaw $k = (n^2 - 3)/2$ kostek. Każda kostka ma kształt prostokąta pokrywającego dokładnie dwa pola. Pola szachownicy są ponumerowane kolejno wierszami – pola w pierwszym wierszu (od lewej do prawej) liczbami od 1 do n, pola w drugim wierszu od $n+1$ do $2n$ i tak dalej aż do pola w prawym dolnym rogu, które ma numer n^2.

Z szachownicy wycinamy trzy dowolne pola, a następnie chcemy ją pokryć kostkami, w taki sposób, aby każde nie wycięte pole pokrywała jedna kostka (pokrywająca też jedno z sąsiednich pól), a pola wycięte pozostały nie pokryte.

Czy zawsze jest to możliwe?

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego SZA.IN rozmiar szachownicy n oraz trzy numery wyciętych pól szachownicy,
- bada, czy taką szachownicę można pokryć kostkami i jeśli nie, to zapisuje w pliku SZA.OUT odpowiedź NIE, a jeśli tak, to zapisuje w pliku SZA.OUT ciąg $k = (n^2 - 3)/2$ par liczb całkowitych wskazujących ułożenie kostek pokrywające daną szachownicę.

Jeśli istnieje wiele sposobów pokrycia szachownicy kostkami, Twój program powinien wypisywać tylko jeden z nich.

Wejście

W jedynym wierszu pliku SZA.IN są zapisane cztery liczby poddzielane pojedynczymi odstępami. Pierwsza liczba to rozmiar szachownicy n, a trzy następne to numery wyciętych pól. Po ostatniej liczbie następuje koniec wiersza.

Dane w pliku SZA.IN są zapisane poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

Plik SZA.OUT powinien zawierać albo jedno słowo NIE, albo w każdym z k kolejnych wierszy dwie liczby oddzielone odstępem, to znaczy numery dwóch sąsiednich pól pokrywanych przez jedną kostkę.

Przykłady

Dla pliku SZA.IN:

7 17 25 40

poprawny plik SZA.OUT może zawierać:

18 19
20 21
8 9
10 11
12 13
14 7
6 5
4 3
1 2
15 22
16 23
29 36
30 37
24 31
32 39
43 44
38 45
46 47
48 49
41 42
33 34
35 28
26 27

Dla pliku SZA.IN:

7 25 32 40

w pliku SZA.OUT powinno być jedno słowo:

NIE

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

Rozwiązanie zadania SZACHOWNICA

Zadanie można zinterpretować i rozwiązywać jako typowe zadanie teoriografowe. Problem sprowadza się do wyznaczenia tzw. pełnego skojarzenia w grafie i ma również związek z problemami istnienia drogi i cyklu Hamiltona. Charakterystycznym dla tego zadania jest jego natura kombinatoryczna – istnienie wielu możliwych przypadków do rozważenia.

Pomalujmy (domyślnie) pola szachownicy kolorami czarnym i białym w zwykły sposób (czyli naprzemiennie w wierszach i w kolumnach) i tak, aby pola narożne były czarne. Jest to możliwe, ponieważ wymiar szachownicy jest nieparzysty. Łatwo zauważyć, że jedna kostka złożona z dwóch połączonych brzegiem pól pokrywa dokładnie jedno białe i jedno czarne pole. Zatem liczba czarnych pól i liczba białych pól do pokrycia muszą być sobie równe, aby zadanie mogło mieć rozwiązanie. Jeśli wymiar szachownicy n jest nieparzysty, to ma ona o jedno czarne pole więcej. Wynika stąd następujące kryterium:

Kryterium. *Szachownicę $n \times n$ (gdzie n jest nieparzyste) z trzema wyciętymi polami można pokryć kostkami wtedy i tylko wtedy, gdy dwa wycięte pola są czarne a jedno białe, lub zgodnie z numeracją w treści zadania, dwa wycięte pola mają numery nieparzyste, a jedno – parzyste.*

Powyżej uzasadniliśmy już konieczność tego warunku, jego dostateczność natomiast jest mniej oczywista i wynika z poprawności podanego poniżej rozwiązania.

Sformułujemy teraz problem z zadania w terminach teorii grafów. Niech $G = (V, E)$ będzie grafem nieskierowanym, gdzie V jest zbiorem **wierzchołków**, a E – zbiorem **krawędzi**, czyli dwuelementowych podzbiorów zbioru wierzchołków. Dwie krawędzie grafu są **niezależne**, jeśli nie mają wspólnych końców. Podzbiór M krawędzi grafu nazywamy **skojarzeniem**, jeśli każde dwie krawędzie w M są niezależne. Skojarzenie w grafie G nazywamy **pełnym**, jeśli każdy wierzchołek grafu należy do którejś krawędzi ze skojarzenia.

Szachownicy $n \times n$ z 3 wyciętymi polami przyporządkujemy graf G, którego zbiór wierzchołków odpowiada niewyciętym polom i dwa pola sąsiadują ze sobą (czyli odpowiadające im wierzchołki w grafie są połączone krawędzią), gdy mają wspólny bok (pionowy lub poziomy). Położenie jednej kostki na szachownicy odpowiada wybraniu jednej krawędzi grafu G, zaś pokrycie kostkami całej szachownicy z wyciętymi polami odpowiada pełnemu skojarzeniu w grafie G. Zatem, zamiast pokrywać szachownicę kostkami możemy szukać pełnego skojarzenia w odpowiadającym jej grafie.

Problem znalezienia skojarzenia w grafie szachownicy sprowadzimy teraz do szukania pewnej dobrej drogi Hamiltona w grafie G', który odpowiada pełnej

szachownicy, czyli bez wyciętych pól. **Drogą Hamiltona** w grafie nazywamy drogę zawierającą każdy wierzchołek grafu dokładnie raz. Droga jest **dobra**, gdy zaczyna się w polu czarnym, oraz wycięte pola leżą na niej w kolejności czarne-białe-czarne (między nimi mogą być inne pola). Oznaczmy wycięte pola przez x, y, z, gdzie x, y są czarne i z jest białe. Jeśli dobra droga Hamiltona w grafie G' ma postać

$$v_1, v_2, \ldots, v_k, x, w_1, w_2, \ldots, w_j, z, u_1, u_2, \ldots, u_r, y, t_1, t_2, \ldots, t_p,$$

to łatwo zauważyć, że k, j, r, p są liczbami parzystymi i można utworzyć pełne skojarzenie w grafie G łącząc w pary kolejne wierzchołki na każdej z częściowych dróg: $v_1, v_2, \ldots, v_k$; $w_1, w_2, \ldots, w_j$; $u_1, u_2, \ldots, u_r$ oraz $t_1, t_2, \ldots, t_p$.

Zatem rozwiązanie zadania sprowadza się teraz do skonstruowania dobrej drogi Hamiltona w grafie G' znając wycięte pola x, y, z (dwa czarne pola x, y i jedno białe pole z).

Poczyńmy najpierw pewne założenia o konfiguracji wyciętych pól na szachownicy. Można założyć, że czarne pola nie znajdują się w tej samej kolumnie. Jeśli tak jest, to możemy zamienić kolumny z wierszami. Można również założyć, że czarne pole x jest wycięte w kolumnie i-tej, a dwa pozostałe pola są wycięte w dalszych kolumnach. Podobnie, jeśli tak nie jest, to tym razem możemy odwrócić numerację kolumn.

Rozważymy dwa przypadki, zależne od konfiguracji pól y i z.

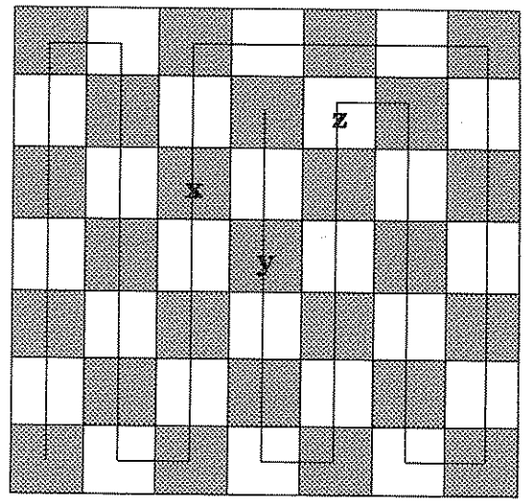
Przypadek 1. Dwa pozostałe wycięte pola (czarne y i białe z) znajdują się w tej samej kolumnie. Rozważmy dwie drogi Hamiltona w grafie G', które rozpoczynają się w dolnym i w górnym polu pierwszej kolumny i przechodzą kolejne pola szachownicy kolumnami. Można łatwo sprawdzić, że jedna z tych dróg jest dobrą drogą Hamiltona.

Przypadek 2. Żadne dwa wycięte pola nie leżą w tej samej kolumnie. Niech i, j, k będą kolumnami, w których są wycięte pola x, y, z. Mamy $i < j$ oraz $i < k$, tzn. czarne pole x jest w kolumnie i, a dwa pozostałe pola, czarne i białe, są w dalszych kolumnach.

Oznaczmy przez $G(p, q)$ graf pełnej szachownicy składający się z kolumn od p-tej do q-tej. Załóżmy, że wycięte czarne pole y nie leży w górnym wierszu szachownicy. Jeśli jest inaczej, to analogicznie możemy rozważać dolny wiersz.

Niech $Path1(i)$ oznacza drogę Hamiltona w grafie $G(1, i)$, która kończy się w górnym polu i-tej kolumny, oraz niech $Path2(y, z, i)$ oznacza drogę Hamiltona w grafie $G(i + 1, n)$, która zaczyna się w górnym polu $(i + 1)$-szej kolumny i pole y występuje w niej po polu z.

Algorytm rozwiązujący zadanie w przypadku 2 składa się z następujących kroków:



Rysunek 1. Droga Hamiltona w drugim przypadku, pola x, z, y rozbijają drogę na 4 parzyste segmenty.

Krok 1. Doprowadź do opisanej powyżej konfiguracji wyciętych pól, poprzez ewentualną zamianę kolumn i wierszy, lub odwrócenie numeracji kolumn lub wierszy.

Krok 2. Skonstruuj drogę $Path1(i)$.

Krok 3. Skonstruuj drogę $Path2(y, z, i)$ – jest to najistotniejsza część algorytmu.

Krok 4. Połącz znalezione powyżej drogi w jedną dobrą drogę Hamiltona.

Krok 5. Wytnij pola x, y, z ze znalezionej drogi Hamiltona i pokryj otrzymane segmenty niezależnymi krawędziami (kostkami).

Jedyną nietrywialną częścią algorytmu jest Krok 3. Pola y i z nie leżą w tej samej kolumnie (obejmuje to już pierwszy przypadek) i można założyć, że pole y nie leży w górnym wierszu. Niech j, k oznaczają kolumny, w których znajdują się pola y, z. Jeśli $k < j$, to drogę $Path2(y, z, i)$ w $G(i + 1, n)$ zaczynamy w górnym polu pierwszej kolumny i obchodzimy tę cześć szachownicy przechodząc kolejne kolumny. W przeciwnym przypadku, przechodzimy pierwszym wierszem do ostatniej kolumny, a potem cofając się przechodzimy kolejne kolumny (pomijając pola pierwszgo wiersza). Odpowiadająca temu droga Hamiltona jest pokazana na rysunku 1 (jest to pierwszy przykład z treści zadania).

W ten sposób w obu przypadkach dobrą drogę Hamiltona w grafie G znajdujemy w czasie liniowym ze względu na całkowity rozmiar szachownicy, tzn. liczbę jej pól.

Rozważmy jeszcze podobne zadania związane z pokryciem szachownicy, w której wycięto dwa lub jedno pole. Jeśli z szchownicy wycięto dwa pola (czarne i białe), to zadanie pokrycia takiej szachownicy kostkami można rozważać dla szachownicy o parzystej liczbie pól. W takim przypadku łatwo skonstruować cykl Hamiltona (tj. drogę zamkniętą przechodzącą przez każde pole dokładnie jeden raz) – wtedy wycięte pola rozdzielają taki cykl na dwie parzystej długości drogi. Oznaczmy algorytm dla tego zadania przez $POKRYCIE_{1,1}$.

Zadanie dla szachownicy z jednym wyciętym polem jest jeszcze prostsze. Szachownica musi mieć nieparzystą liczbę pól. Niech narożne pola będą koloru czarnego, wtedy wyciąć możemy pole czarne. Wybieramy dowolną drogę Hamiltona. Wycięte czarne pole rozcina tę drogę na parzyste części. Oznaczmy algorytm dla tego przypadku przez $POKRYCIE_1$.

Zauważmy, że w przypadku wycinania z szachownicy jednego lub dwóch pól, rozważany cykl i droga Hamiltona nie zależą od rozmieszczenia wyciętych pól.

Oznaczmy przez $POKRYCIE_{1,2}$ algorytm, który znajduje pokrycie szachownicy z naszego zadania konkursowego, czyli z wyciętymi jednym polem białym i dwoma polami czarnymi. Podaliśmy już jeden taki algorytm, a teraz podamy algorytm rekurencyjny, który korzysta z algorytmów pokrywających szachownicę z jednym lub dwoma polami wyciętymi.

Niech i, j, k oznaczają kolumny, w których zostały wycięte pola x, y, z, gdzie x, y są czarne i z jest białe. Tak jak poprzednio, możemy założyć, że $i < j$ oraz $i < k$, tzn. czarne pole jest w kolumnie i, a pola czarne i białe są w kolumnach dalszych niż i. Jeśli i jest nieparzyste, to stosujemy algorytm $POKRYCIE_1$ do pierwszych i kolumn szachownicy i algorytm $POKRYCIE_{1,1}$ do pozostałych kolumn. W przeciwnym przypadku, pokrywamy jedną kostką dwa sąsiednie niewycięte pola, jedno białe w i tej kolumnie, i jedno czarne w $(i + 1)$-szej kolumnie. Potraktujmy te pola jako wycięte. Teraz możemy zastosować algorytm $POKRYCIE_{1,1}$ do pierwszych i kolumn, oraz (rekurencyjnie) algorytm $POKRYCIE_{1,2}$ do pozostałych kolumn.

W tym podejściu nie można dopuścić do sytuacji, gdy pozostaje tylko jedna (ostatnia) kolumna, z wyciętymi trzema polami – takiej kolumny może się nie udać pokryć kostkami. Rekurencję należy zatem zatrzymać w bezpiecznym miejscu lub w momencie, gdy wycięte pole czarne i białe są w tej samej kolumnie, podczas gdy drugie czarne jest w innej. Wtedy można postąpić podobnie jak w pierwszym przypadku w algorytmie opisanym na początku.

Przeprowadzona analiza istnienia i postaci rozwiązania zadania dostarcza gotowego przepisu, w jaki sposób pokrywać szachownicę kostkami, jeśli spełniony jest warunek kryterium, podanego na początku. Z tego względu nie zamieszczamy żadnego fragmentu kodu, gdyż napisanie programu generującego

istniejące pokrycie na podstawie opisanych powyżej dokładnie konfiguracji drogi Hamiltona w grafie odpowiadającym szachownicy jest proste i nie powinno sprawić większego kłopotu.

Omówienie rozwiązań podanych przez uczniów

Większość uczniów odkryła warunek konieczny na istnienie pokrycia (sformułowany przez nas w kryterium). W programach natomiast stosowano rozmaite heurystyki dla wykazania jego dostateczności, a więc nieformalne metody, które miały eliminować rozpatrywanie zbyt dużej liczby możliwości.

Jedną z metod było usuwanie z szachownicy zewnętrznej ramki, utworzonej przez nie wycięte pola, do momentu pojawienia się wyciętego pola na brzegu pozostałej części szachownicy.

Innym sposobem było pokrywanie kostkami najpierw przymusowych konfiguracji w szachownicy (np. dwóch pól zamkniętych przez wycięte pola), a potem pozostałych części, w których było wiele możliwości wyboru.

Stosowano również metodę poszukiwania z nawrotami (ang. *backtracking*), często realizując ją z wykorzystaniem rekurencji. Niektóre takie rozwiązania korzystały z kryterium istnienia, a niektóre po prostu próbowały znaleźć pokrycie szachownicy kostkami biorąc pod uwagę niemal wszystkie ustawienia klocków. W przeciwieństwie do innych rozwiązań, programy realizujące przeszukiwanie z nawrotami przekraczały zwykle limity czasowe przydzielone poszczególnym testom.

Niektóre ze stosowanych metod działały poprawnie dla danych testowych, ale dowód ich ogólnej poprawności był niejasny.

Testy

Pod względem rozmiaru były dwie grupy testów: dwie szachownice o rozmiarach 7×7 (wśród nich przykład z treści zadania, zob. rys. 1), jedna 3×3 i 20 szachownic o rozmiarach 49×49.

Testy z tej pierwszej grupy miały na celu przede wszystkim zbadanie poprawności generowanych rozwiązań, a z tej drugiej – były ponadto testami sprawdzającymi efektywność zaprogramowanych algorytmów. Wśród wszystkich testów, 17 szachownic dawało się pokryć kostkami, a 6 – nie. Wycięte pola były albo skupione, albo porozrzucane po całej szachownicy. W obu przypadkach, w części testów wycięte pola znajdowały się przy brzegu szachownicy. Konfiguracje wyciętych pól z szachownicy były tak dobrane, by każda z metod opisanych w poprzednim punkcie (jury przewidziało pojawienie się prawie wszystkich typów rozwiązań uczniowskich) miała utrudnione działanie.

Podajemy dane testowe w postaci, w jakiej były umieszczone w pliku SZA.IN:

```
7 17 25 40
7 41 47 42
3 1 5 9
49 980 1028 1078
49 980 1029 1078
49 1325 1422 1373
49 1371 1470 1421
49 1197 1687 1198
49 1246 1295 1247
49 2158 2255 2353
49 2172 593 1411
49 628 713 2119
49 480 2073 1392
49 2169 467 1855
49 520 2320 988
49 3 4 53
49 47 46 95
49 2355 2356 2307
49 2399 2398 2349
49 99 148 149
49 2255 2206 2207
49 147 196 195
49 2303 2254 2253
```

7.2.5. Zadanie WIELOKĄT (Autor: Krzysztof Diks)

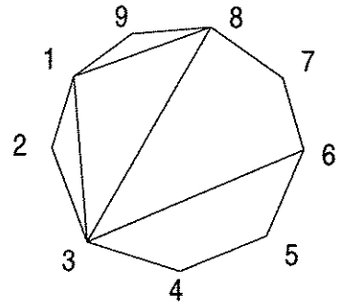
Treść zadania WIELOKĄT

Dany jest n-kąt wypukły W (gdzie $3 < n \leq 5000$) i k jego różnych przekątnych, parami nie przecinających się wewnątrz wielokąta. (Jedynym wspólnym punktem dwóch różnych przekątnych może być tylko wierzchołek wielokąta.)* Wierzchołki wielokąta są ponumerowane kolejno od 1 do n w kierunku przeciwnym do ruchu wskazówek zegara. Wszystkie przekątne dzielą W na mniejsze wielokąty wypukłe o rozłącznych wnętrzach.

* Uwaga o tym, że przekątne mogą mieć wspólne punkty tylko w wierzchołkach wielokąta została podana na początku zawodów.

Przykład

Cztery przekątne 1–8, 8–3, 3–1 i 3–6 dzielą wielokąt W przedstawiony na poniższym rysunku na dwa czworokąty i trzy trójkąty.



Zadanie

Ułóż program, który:

- wczytuje opis wielokąta W i jego przekątnych z pliku tekstowego WIE.IN;
- oblicza maksymalną liczbę boków wielokąta wśród wielokątów wypukłych powstałych z podziału W;
- zapisuje wynik w pliku tesktowym WIE.OUT.

Wejście

W każdym wierszu pliku tekstowego WIE.IN są zapisane dwie liczby całkowite dodatnie oddzielone pojedynczym odstępem.

W pierwszym wierszu jest zapisana liczba wierzchołków wielokąta n i liczba przekątnych k.

W każdym z kolejnych k wierszy znajduje się opis jednej przekątnej wielokąta w postaci pary liczb całkowitych dodatnich – numerów wierzchołków, które łączy ta przekątna; bezpośrednio po drugiej z liczb następuje koniec wiersza.

Dane w pliku WIE.IN są zapisane poprawnie i Twój program nie musi tego sprawdzać.

Wyjście

W pliku WIE.OUT należy zapisać jedną liczbę całkowitą dodatnią – maksymalną liczbę boków wielokąta wypukłego powstałego z podziału danego wielokąta W.

Przykład

Dla następującego pliku WIE.IN, który jest opisem wielokąta przedstawionego na rysunku:

```
9 4
1 8
8 3
3 1
3 6
```

w pliku WIE.OUT powinna być zapisana jedna liczba:

```
4
```

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

Rozwiązanie zadania WIELOKĄT

Oznaczmy przez P zbiór przekątnych wielokąta W danych w pliku wejściowym WIE.IN. Każdą przekątną będziemy jednoznacznie reprezentowali przez uporządkowaną parę liczb całkowitych (a, b), gdzie a, b są jej końcami (wierzchołkami wielokąta) oraz $a < b$. Zgodnie z treścią zadania, przekątne dzielą wielokąt W na mniejsze wielokąty wypukłe bez przekątnych. Nazwijmy każdy taki wielokąt **ścianą**. Ścianę można jednoznacznie opisać podając kolejno wszystkie jej wierzchołki w porządku rosnącym. Ścianami wielokąta z przykładu są:

1, 2, 3; 1, 3, 8; 1, 8, 9; 3, 6, 7, 8; 3, 4, 5, 6.

Mówimy, że ściana S jest **maksymalna**, jeżeli liczba jej boków nie jest mniejsza niż liczba boków każdej innej ściany. Naszym celem jest napisanie programu znajdującego liczbę boków ściany maksymalnej. Oznaczmy tę liczbę przez $MaxW$. Dla wielokąta z przykładu $MaxW = 4$. Niech (a, b) będzie przekątną ze zbioru P lub bokiem $(1, n)$. Oznaczmy przez $W(a, b)$ wielokąt zawarty w wielokącie W o wierzchołkach $a, a+1, \ldots, b$ (w szczególności $W(1, n) = W$), a przez $P(a, b)$ – zbiór tych przekątnych z P, które są przekątnymi wielokąta $W(a, b)$. Dla wielokąta z przykładu mamy $P(1, 8) = \{(1, 3), (3, 8), (3, 6)\}$. **Ścianą główną** w wielokącie $W(a, b)$ nazwiemy ścianę w tym wielokącie, której jednym z boków jest (a, b). Dla wielokąta z przykładu, ścianą główną wielokąta $W(3, 8)$ jest ściana 3, 6, 7, 8. Niech (a_1, b_1), (a_2, b_2), ..., (a_m, b_m) będę przekątnymi ze

zbioru $P(a, b)$, które są jednocześnie bokami ściany głównej w wielokącie $W(a, b)$, gdzie $a_1 < a_2 < \ldots < a_m$ – przekątne te nazwiemy głównymi w wielokącie $W(a, b)$. Zauważmy, że liczba boków ściany głównej wielokąta $W(a, b)$ wynosi

$$(b - a + 1) - (\sum_{l=1}^{m} (b_l - a_l - 1). \qquad (1)$$

Oznaczmy tę liczbę przez $LB(a, b)$. W naszym przykładzie mamy $LB(3, 8) = (8 - 3 + 1) - (6 - 3 - 1) = 4$.

Niech $MaxW(a, b)$ oznacza liczbę boków maksymalnej ściany w wielokącie $W(a, b)$. $MaxW(a, b)$ można wyrazić rekurencyjnie jak następuje:

$$MaxW(a, b) = \max\{LB(a, b), MaxW(a_1, b_1), \ldots, MaxW(a_m, b_m)\}.$$

Dla wielokąta z przykładu

$$MaxW(1, 8) = \max\{LB(1, 8) = 3, MaxW(1, 3) = 3, MaxW(3, 8) = 4)\} = 4.$$

Aby rozwiązać zadanie wystarczy obliczyć $MaxW(1, n)$.

Pokażemy teraz w jaki sposób można zrobić to elegancko i jednocześnie szybko (efektywnie). Do tego celu jest potrzebna odpowiednia reprezentacja wielokąta, która umożliwi łatwą identyfikację wszystkich jego ścian. W proponowanym rozwiązaniu wykorzystamy fakt, że w wielokącie $W(a, b)$ wielokąty $W(a_1, b_1)$, $W(a_2, b_2)$, ..., $W(a_m, b_m)$ mają rozłączne wnętrza.

Uporządkujmy przekątne tak, żeby przekątna (a, b) poprzedzała przekątną (c, d) wtedy i tylko wtedy, gdy

$$a < c \text{ lub } (a = c \text{ i } b > d), \qquad (2)$$

co będziemy zapisywali $(a, b) [(c, d)$.

Dla wielokąta z przykładu kolejność przekątnych po uporządkowaniu jest następująca: (1, 8), (1, 3), (3, 8), (3, 6).

Jeżeli wielokąt $W(a, b)$ ma przekątne w zbiorze P (tzn. zbiór $P(a, b)$ jest niepusty), to wszystkie jego przekątne występują w porządku [kolejno, poczynając od przekątnej występującej bezpośrednio po (a, b). Co więcej, jeżeli (a_1, b_1), ..., (a_m, b_m) są przekątnymi głównymi $W(a, b)$, dla $a_1 < \ldots < a_m$, to bezpośrednio po (a, b) występuje przekątna (a_1, b_1), potem przekątne wielokąta $W(a_1, b_1)$, następnie przekątna (a_2, b_2) i po niej przekątne z $W(a_2, b_2)$, na koniec zaś przekątna (a_m, b_m) i po niej przekątne wielokąta $W(a_m, b_m)$.

Niech $(a(Poz), b(Poz))$ będzie przekątną na pozycji Poz w porządku [. Dla wielokąta z przykładu mamy $a(3) = 3$, $b(3) = 8$. Dotychczasowe rozważania umożliwiają obliczanie $MaxW = MaxW(1, n)$ za pomocą procedury `ObejrzyjWie` zdefiniowanej poniżej, która rekurencyjnie przegląda ściany wielokąta $W(a(Poz), b(Poz))$ i aktualizuje wartość zmiennej `MaxW`, jeśli właśnie obejrzana

ściana ma większą liczbę boków. W procedurze ObejrzyjWie przekątne główne wielokąta $W(a(Poz), b(Poz))$ są przeglądane w porządku zgodnym z relacją [. Jednocześnie, zgodnie ze wzorem (1), jest zliczana liczba boków ściany głównej tego wielokąta. Dla każdej głównej przekątnej (c, d) (a raczej dla wielokąta $W(c, d)$) jest rekurencyjnie wywoływana procedura ObejrzyjWie. W momencie wywołania procedury ObejrzyjWie wartością Poz jest pozycja przekątnej (c, d). Przeglądanie ścian wielokąta W rozpoczyna się od jego ściany głównej. Dlatego przyjmujemy, że $a(0) = 1$ i $b(0) = n$. Dodatkowo zakładamy, że $a(k + 1) = n$.

Poniżej zamieszczamy deklaracje odpowiednich struktur danych oraz procedurę ObliczMaxW, która wykorzystuje procedurę ObejrzyjWie i wyznacza liczbę boków w maksymalnej ścianie wielokąta W.

```
const
 MaxN=5000;    {Maksymalna liczba wierzcholkow w
                wielokacie.}
 MaxP=MaxN-2; {Maksymalna liczba przekatnych w
                wielokacie+1.}
 {Przekatne reprezentujemy jako pare liczb.}
type PrzekatnaTyp=record
                    a,b:Integer
                  end;

var Przekatne:array[0..MaxP] of PrzekatnaTyp;

 {Tablica przekatnych. Kazda przekatna jest pamietana jako
  uporzadkowana rosnaco para liczb. Przekatne w tablicy
  zostana posortowane w porzadku [ zdefiniowanym w (2). Po
  uporzadkowaniu wszystkie przekatne w wielokacie W(a,b)
  wystepuja kolejno, poczynajac od przekatnej bezposrednio
  wystepujacej po (a,b). Przyjmujemy, ze Przekatne[0]
  reprezentuje bok (1,n) oraz Przekatna[k+1].a=n+1 -
  przekatna ta odgrywa role wartownika w petli while w
  procedurze ObejrzyjWie.}

procedure ObliczMaxW;
  {Wyznaczanie liczby bokow w maksymalnej scianie wielokata
   W.}
 var MaxW,Poz:Integer;
```



```
  {Zmienna MaxW kumuluje liczbe bokow najwiekszej sciany.
   Poz jest indeksem w tablicy Przekatne, aktualnie
   przetwarzanej przekatnej.}

 procedure ObejrzyjWie;
  {Procedura ObejrzyjWie przeglada rekurencyjnie sciany
   wielokata W(Przekatne[Poz].a,Przekatne[Poz].b), zlicza
   ich boki i aktualizuje MaxW - liczbe bokow w
   najwiekszej z dotychczas obejrzanych scian w calym
   wielokacie W.}
 var
  LB,                    {Liczba bokow sciany glownej wielokata
                          W(Przekatne[Poz].a,Przekatne[Poz].b).}
  PrawyKoniec:Integer; {PrawyKoniec - zmienna pomocnicze}
 begin {ObejrzyjWie}
  LB:=Przekatne[Poz].b-Przekatne[Poz].a+1;
  PrawyKoniec:=Przekatne[Poz].b;
  Poz:=Poz+1;
  while Przekatne[Poz].a<PrawyKoniec do begin
   LB:=LB-(Przekatne[Poz].b-Przekatne[Poz].a)+1;
   ObejrzyjWie
  end;
  if LB>MaxW then MaxW:=LB
 end; {ObejrzyjWie}

begin {ObliczMaxW}
 MaxW:=0; Poz:=0;
 ObejrzyjWie {Przejrzyj wszystkie sciany wielokata W.}
 {MaxW jest liczba bokow w najwiekszej scianie wielokata W.}
end; {ObliczMaxW}
```

Wyznaczmy teraz jaka jest złożoność obliczania *MaxW*. Nietrudno zauważyć, że liczba wszystkich operacji w naszym algorytmie jest proporcjonalna do łącznej liczby wywołań procedury ObejrzyjWie. Procedura ObejrzyjWie jest wywoływana dokładnie raz dla każdej przekątnej oraz boku $(1, n)$. Stąd, jeżeli przekątne wielokąta są uporządkowane zgodnie z relacją [, to wyznaczenie maksymalnej liczby boków w ścianie n-kąta z k przekątnymi zabiera czas $O(k)$. Zauważmy, że czas ten zależy tylko od liczby przekątnych, co jest cenne

w sytuacji gdy n jest bardzo duże, a k małe. W najgorszym przypadku jednak n-kąt może mieć $k = n - 3$ przekątnych.

Pozostaje pokazać w jaki sposób uporządkować przekątne wielokąta zgodnie z relacją [. Do tego celu można użyć dowolnego algorytmu sortowania, np. algorytmu sortowania stogowego (*HeapSort*). Opis tej metody i jej własności można znaleźć w wielu książkach poświęconych algorytmom, zob. np. L. Banachowski, A. Kreczmar, *Elementy analizy algorytmów*, WNT, Warszawa 1982. Aby zaadaptować dla naszych celów realizację algorytmu stogowego podaną tamże (str. 108) wystarczy dla podanego typu danych, reprezentującego przekątne, zastąpić relację $a < b$ przez następującą funkcję `Mniejsza` porównującą przekątne według relacji zdefiniowanej w (2).

```
function Mniejsza(i,j:Integer):Boolean;
 {Porownywanie przekatnych zgodnie z relacja [. Mniejsza jest
  wartoscia relacji Przekatne[i] [ Przekatne[j].}
begin
  Mniejsza:=(Przekatne[i].a<Przekatne[j].a)
                          or
           ((Przekatne[i].a=Przekatne[j].a) and
            (Przekatne[i].b>Przekatne[j].b))
end; {Mniejsza}
```

Za pomocą algorytmu sortowania stogowego k przekątnych można uporządkować w czasie $O(k\log k)$. Wynika stąd, że złożoność zaprezentowanego algorytmu wyznaczania liczby boków w maksymalnej ścianie wielokąta zawierającego k przekątnych wynosi $O(k\log k)$.

Omówienie rozwiązań podanych przez uczniów

Zadanie WIELOKĄT sprawiło zawodnikom dużo kłopotów. Główną trudnością, z którą nie mogli oni sobie poradzić była właściwa reprezentacja wielokąta, umożliwiająca identyfikację i systematyczne przeglądanie jego ścian. Tylko kilkunastu uczniów w pełni poradziło sobie z tym problemem.

Wśród poprawnych metod rozwiązania zadania można wyróżnić trzy.

1. W metodzie 1 ściany wielokąta obchodzi się w kierunku przeciwnym do ruchu wskazówek zegara. W tym celu, dla każdego wierzchołka v jest pamiętana uporządkowana lista boków i przekątnych o początku w v. Przekątne i boki na każdej takiej liście występują w kolejności zgodnej z kolejnością występowania na obwodzie wielokąta, w kierunku przeciwnym do ruchu wskazówek zegara, wierzchołków do których prowadzą.

Każda przekątna (u, v) jest bokiem dokładnie dwóch ścian, a każdy bok wielokąta jest bokiem dokładnie jednej ściany. Rozważmy przekątną (u, v), gdzie $u < v$. Przekątna ta dzieli wielokąt W na dwa wielokąty W_1 i W_2. Wielokąt W_1 zawiera wierzchołki od v do u, a wielokąt W_2 od u do v. Jedna ze ścian W, której bokiem jest (u, v) leży w W_1, druga natomiast w W_2. Oznaczmy te ściany, odpowiednio, przez S_1 i S_2. Zauważmy teraz, że kolejnym bokiem po (u, v) na ścianie S_1, w kierunku przeciwnym do ruchu wskazówek zegara, jest przekątna (lub bok) występująca bezpośrednio przed (u, v) na liście wierzchołka v. Natomiast kolejnym bokiem po (u, v) w ścianie S_2 jest przekątna (lub bok) bezpośrednio poprzedzająca (u, v) na liście u. Podobnie, dla każdego boku (u, v) wielokąta W można określić, który bok lub przekątna będzie kolejnym bokiem w jedynej ściany zawierającej (u, v). Jeżeli dla każdej przekątnej (lub boku) (u, v) na liście dla u, mamy wskaźnik do wystąpienia (u, v) na liście v i odwrotnie, dla (u, v) na liście v, mamy wskaźnik do wystąpienia (u, v) na liście u, to każdą ze ścian S_1 i S_2 możemy obejść w czasie proporcjonalnym do liczby boków tej ściany. Stąd, żeby wyznaczyć maksymalną liczbę boków w ścianie wielokąta W wystarczy dla każdej przekątnej przejrzeć obie ściany, których jest ona bokiem. Żeby uniknąć wielokrotnego przeglądania tych samych ścian można wprowadzić dla każdej przekątnej wskaźnik informujący, która ze ścian zawierająca tę przekątną jako bok, nie była dotychczas przeglądana. Wynika stąd, że wszystkie ściany możemy przejrzeć w czasie proporcjonalnym do sumy ich długości (liczonej liczbą boków), która wynosi $n + 2k$. Żeby zbudować opisaną reprezentację, dla każdej wejściowej pary u, v opisującej przekątną, tworzy się dwie uporządkowane pary (u, v) oraz (v, u). Następnie, wszystkie uporządkowane pary sortuje się w taki sposób, żeby pary o tej samej pierwszej składowej, tzn. przekątne wychodzące z tego samego wierzchołka, występowały w porządku opisanym wcześniej. Ponieważ sortowanie można wykonać w czasie $O(k\log k)$, złożoność tej metody wynosi $O(k\log k + n)$.

2. Nazwijmy długością przekątnej (a, b), gdzie $a < b$, liczbę

$$\min\{b - a + 1, n - (b - a - 1)\}.$$

Długość przekątnej jest równa liczbie wierzchołków mniejszego z wielokątów powstałych z W w wyniku podzielenia go przekątną (a, b). W metodzie 2 najpierw przekątne są porządkowane niemalejąco według ich długości. Niech $(a_1, b_1), \ldots, (a_k, b_k)$ będą kolejnymi przekątnymi w tym porządku. W celu przejrzenia ścian wielokąta W konstruuje się ciąg wielokątów $W_1, \ldots, W_k$, taki że:

— $W_1 = W$,
— dla każdego $i = 1, \ldots, k-1$, W_{i+1} otrzymuje się z W_i przez odcięcie mniejszego z wielokątów, powstających w wyniku podziału W_i przekątną (a_i, b_i).

Zauważmy, że każdy odcinany wielokąt jest ścianą w W. Jeżeli wielokąt W_i jest pamiętany na liście cyklicznej, na której wierzchołki występują w kolejności przeciwnej do ruchu wskazówek zegara, to odcięcie można wykonać w czasie stałym, a następnie wyznaczyć liczbę boków odcinanego wielokąta w czasie proporcjonalnym do tej liczby. Złożoność tej metody jest taka sama jak metody poprzedniej i wynosi $O(k \log k + n)$.

3. W metodzie 3 stosowano zasadę dziel i zwyciężaj. Spośród przekątnych jest wybierana dowolna przekątna (a, b). Przekątna (a, b) dzieli W na dwa wielokąty W_l i W_p. Żeby znaleźć maksymalną liczbę boków w ścianie w wielokącie W, należy podzielić przekątne na te, które są przekątnymi w W_l i te, które są przekątnymi w W_p, a następnie rekurencyjnie obliczyć maksymalne liczby boków w ścianach wielokątów W_l i W_p. Wieksza z tych liczb jest szukaną maksymalną liczbą boków. Problemem pojawiającym się przy realizacji tej metody jest zapamiętywanie wielokątów pojawiających się w wyniku podziału. Niestety, wierzchołki tych wielokątów nie muszą być kolejnymi liczbami. Nasuwający się sposób pamiętania wielokątów, to listy cykliczne jego wierzchołków. Wtedy podziału można dokonać w czasie stałym. Wielokąt powstający w wyniku podziałów i nie zawierający przekątnych, jest ścianą wejściowego wielokąta W. Żeby policzyć boki ściany wystarczy policzyć wierzchołki na liście reprezentującej ten wielokąt. Łącznie dla wszystkich ścian zabiera to czas $O(n)$.

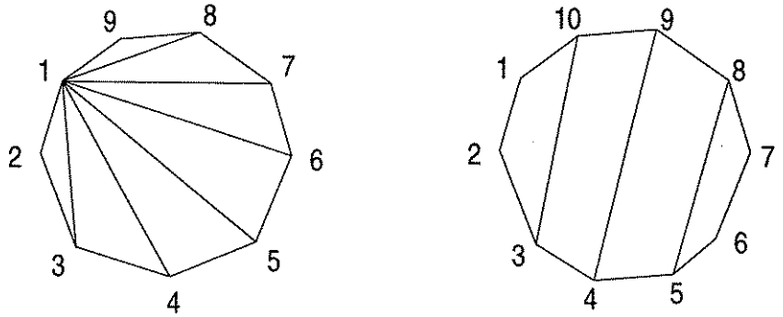
Złożoność trzeciej metody zależy również od wyboru przekątnej do podziału wielokąta. Pechowo może się zdarzać, że wszystkie przekątne będą wpadać zawsze do tylko jednego z wielokątów W_l lub W_p. Aby stwierdzić, do którego wielokąta wpada przekątna trzeba porównać ją z przekątną dzielącą. W najgorszym przypadku wykonuje się najpierw $k-1$ takich porównań, potem $k-2$, następnie $k-3$ itd. Na końcu mamy 1 porównanie. Łatwo obliczyć, że łączna liczba porównań wynosi $k(k-1)/2$. W praktyce, dla przeciętnych danych, łączna liczba porównań jest zazwyczaj dużo mniejsza.

Podsumowując, złożoność metody 3, w najgorszym przypadku wynosi $O(n + k^2)$.

Testy

Do sprawdzania rozwiązań zawodników użyto 8 testów. Dwa spośród nich sprawdzały, w jaki sposób program ucznia radzi sobie z przypadkami brzegowymi: tzn. z wielokątami, odpowiednio, o 1 i maksymalnej możliwej liczbie

przekątnych. Pozostałe testy, oprócz weryfikacji poprawności rozwiązań, służyły także do sprawdzenia szybkości działania rozwiązań. We wszystkich tych testach, poza jednym, dane wielokąty miały rozmiar 5000. (W jednym przypadku liczba wierzchołków wielokąta wejściowego wynosiła 2000). Liczby przekątnych wahały się od 1996 do 4996. Wielokąty wejściowe miały bardzo różnorodną strukturę, od wielokąta w którym wszystkie przekątne wychodziły tylko z jednego wierzchołka, do wielokąta w którym z każdego wierzchołka wychodziła co najwyżej jedna przekątna (zob. rys. poniżej).



7.3. Zawody III stopnia

7.3.1. Zadanie KODOWANIE PERMUTACJI (Autor: Krzysztof Diks)

Treść zadania KODOWANIE PERMUTACJI

Każdą permutację $\alpha = (a_1, \ldots, a_n)$ liczb $1, \ldots, n$ można zakodować za pomocą ciągu $\beta = (b_1, \ldots, b_n)$, w którym b_i jest równe liczbie wszystkich a_j takich, że: $(j < i \ \& \ a_j > a_i)$, dla każdego $i = 1, \ldots, n$.

Przykład

Kodem permutacji $\alpha = (1, 5, 2, 6, 4, 7, 3)$ jest ciąg $\beta = (0, 0, 1, 0, 2, 0, 4)$.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego KOD.IN długość n i kolejne wyrazy ciągu liczb β,
- sprawdza, czy jest on kodem jakiejś permutacji liczb $1, \ldots, n$,

- jeżeli tak, to znajduje tę permutację i zapisuje ją w pliku tekstowym KOD.OUT,
- w przeciwnym przypadku zapisuje w pliku KOD.OUT jedno słowo NIE.

Wejście

- W pierwszym wierszu pliku tekstowego KOD.IN jest zapisana dodatnia liczba całkowita $n \leq 30000$. Jest to liczba wyrazów ciągu β.
- W każdym z kolejnych n wierszy jest zapisana jedna liczba całkowita nieujemna nie większa niż 30000. Jest to kolejny wyraz ciągu β.

Wyjście

W pliku tekstowym KOD.OUT należy zapisać:

- w każdym z kolejnych n wierszy jeden wyraz permutacji α, której kodem jest dany ciąg β, zapisany w pliku KOD.IN,
- jedno słowo NIE, jeśli ciąg β nie jest kodem żadnej permutacji.

Przykłady

Dla pliku KOD.IN:

```
7
0
0
1
0
2
0
4
```

w pliku KOD.OUT należy zapisać:

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

Dla pliku KOD.IN:

```
4
0
2
0
0
```


w pliku KOD.OUT należy zapisać:

NIE

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

Rozwiązanie zadania KODOWANIE PERMUTACJI

Sformułujmy najpierw warunek rozstrzygający, czy ciąg $\beta = (b_1, \ldots, b_n)$ jest kodem permutacji liczb $(1, \ldots, n)$. Zauważmy, że jeżeli β jest kodem permutacji, to

$$0 \leq b_i < i, \text{ dla każdego } i = 1, \ldots, n. \tag{1}$$

Z drugiej strony pokażemy, że jeśli $\alpha_1 = (a_1^1, a_2^1, \ldots, a_n^1)$ i $\alpha_2 = (a_1^2, a_2^2, \ldots, a_n^2)$ są dwiema różnymi permutacjami, to ich kody, odpowiednio, β_1 i β_2, są również różne. W tym celu rozważmy największe i takie, że $a_i^1 \neq a_i^2$. Bez utraty ogólności możemy przyjąć, że $a_i^1 > a_i^2$. Ponieważ zbiory elementów leżących na lewo od a_i^1 i a_i^2 w permutacjach, odpowiednio, α_1 i α_2 różnią się tylko elementami a_i^1 oraz a_i^2, więc $b_i^1 < b_i^2$. Zatem różnych kodów permutacji jest $n!$. Z drugiej strony, różnych ciągów $(b_1, \ldots, b_n)$ spełniających (1) jest także $n!$. Stąd wynika, że warunek (1) jest warunkiem koniecznym i dostatecznym na to, aby ciąg β był kodem pewnej, dokładnie jednej permutacji.

Załóżmy, że ciąg β spełnia warunek (1). Pokażemy teraz w jaki sposób można znaleźć permutację, której kodem jest β. Nasze rozważania będziemy ilustrować przykładem ciągu $\beta = (0, 0, 1, 0, 2, 0, 4)$ z treści zadania. Permutację $\alpha = (a_1, \ldots, a_7)$, której kodem jest β budujemy od końca. W tym celu rozważmy ciąg $q = [7, 6, 5, 4, 3, 2, 1]$. Zauważmy, że ponieważ $b_7 = 4$, to na lewo od a_7 w permutacji α muszą być dokładnie 4 elementy większe. Jedynym elementem spełniającym ten warunek jest 3. Czyli $a_7 = 3$. Usuńmy 3 z q. Ciąg q ma teraz postać $[7, 6, 5, 4, 2, 1]$. Przystępujemy do obliczania a_6. Ponieważ $b_6 = 0$, więc a_6 jest największym elementem wśród elementów q. Stąd $a_6 = 7$. Usuńmy 7 z q. Teraz mamy $q = [6, 5, 4, 2, 1]$. Obliczamy a_5. Ponieważ $b_5 = 2$, więc a_5 będzie równe trzeciemu elementowi ciągu q, czyli $a_5 = 4$. Usuwamy 4 z q i powtarzamy to postępowanie jeszcze 4 razy obliczając kolejno $a_4, \ldots, a_1$.

Można to zapisać bardziej formalnie w następujący sposób.

Algorytm Permutacja.

$q := [n, n-1, \ldots, 1]$.

Dla $i = n, n-1, \ldots, 1$ wykonaj:

Niech $k := b_i + 1$;

Przyjmij za a_i k-ty element ciągu q;

Usuń a_i z q.

Aby zaprogramować ten algorytm musimy najpierw podać, w jaki sposób będą reprezentowane w nim ciąg q i jak znajdować w nim k-ty element, a następnie usuwać go z tego ciągu. Dwoma najprostszymi reprezentacjami ciągu q są reprezentacje tablicowa i listowa.

W reprezentacji tablicowej, ciąg q jest pamiętany w tablicy zero-jedynkowej Q [1..n], gdzie Q [i] = 1 wtedy i tylko wtedy, gdy liczba i jest elementem tego ciągu. Znalezienie k-tego elementu w ciągu q polega na przejrzeniu tablicy Q od strony prawej do lewej (tzn. zgodnie z malejącymi indeksami) i znalezieniu takiego i, że Q [i] = 1 oraz dla dokładnie k–1 indeksów $j > i$ jest Q [j] = 1. Usunięcie znalezionego elementu z ciągu polega na przypisaniu $Q[i]$ wartości 0.

W reprezentacji listowej ciąg q jest pamiętany jako uporządkowana malejąco lista jego elementów. Wyszukanie k-tego elementu w tym ciągu polega na pominięciu k–1 pierwszych elementów listy. Usunięciu elementu z ciągu q odpowiada usunięcie go z listy.

Zauważmy, że w obu reprezentacjach łączna liczba operacji na ciągu (sprawdzanie, czy Q [i] = 1 lub przejście przez elementy listy) jest nie mniejsza niż $\sum_{i=1}^{n} b_i$. Ponieważ suma ta może przyjmować wartości między 0 a $n(n-1)/2$ włącznie, to dla dużych n (w naszym zadaniu n może być równe 30000) i pewnych kodów takie realizacje algorytmu mogą być bardzo czasochłonne.

Zaproponujemy dużo efektywniejszą metodę i do tego równie prostą. Dla liczb całkowitych $1 \le l \le p \le n$, niech $q[l, p]$ oznacza podciąg kolejnych elementów ciągu q z przedziału domkniętego $[l, p]$. Na przykład dla $q = [7, 5, 3, 2, 1]$, mamy $q[2, 4] = [3, 2]$. K-ty element w ciągu q znajdujemy za pomocą procedury `SzukajUsun`, umieszczonej poniżej w procedurze `Permutacja`, która realizuje wyżej opisany algorytm o tej samej nazwie. W programie, który tworzymy, korzystamy z następującej stałej i typów danych:

```
const
 Nmax=30000;
type
 WskDoTablicy=^Tablica;
  {Uzywamy tablic dynamicznych, gdyz w TP pamiec przeznaczona
```



```
   na zmienne statyczne jest ograniczona do 65520 bajtow.}
 Tablica      =array[1..Nmax] of Word;
```

A oto opis procedury `Permutacje`. Znaczenie tablicy `wPrzedz` oraz instrukcji znajdujących się w wierszach oznaczonych na początku przez {*} jest wyjaśnione w dalszej treści.

```
procedure Permutacja(B,wPrzedz:WskDoTablicy; DlugB:Word);
 {W procedurze tej jest obliczana permutacja, ktorej kodem
  jest ciag o dlugosci DlugB zapisany w tablicy B^.
  Permutacja ta jest wyznaczana za pomoca tablicy wPrzedz^.
  Obliczana permutacja jest zapamietywana w tablicy B^.}
function SzukajUsun(k:Word):Word;
 {Wartoscia funkcji jest k-ty element w ciagu q - funkcja
  usuwa go z tego ciagu i aktualizuje tablice wPrzedz^.}
 var
  Lewy,Prawy,Srodek,Ile,naLewo,naPrawo,
  r,pom                                    :Word;
  Znaleziono                               :Boolean;
begin
 Lewy:=1;  Prawy:=DlugB;
 r:=k;  Znaleziono:=False;
 repeat
  {k-ty element w ciagu q jest r-tym elementem w ciagu
   q[Lewy,Prawy]}
  Srodek:=Lewy+(Prawy-Lewy) div 2;
  {Ile jest liczba elementow w ciagu q[Lewy,Prawy],
   naLewo jest liczba elementow ciagu q z przedzialu
   [Lewy,Srodek-1], naPrawo jest liczba elementow ciagu q
   z przedzialu [Srodek+1,Prawy].}
{*}    Ile:=wPrzedz^[Srodek];
{*}    if Lewy<Srodek then
{*}     naLewo:=wPrzedz^[(Lewy+Srodek-1) div 2]
{*}    else naLewo:=0;
{*}    if Prawy>Srodek then
{*}     naPrawo:=wPrzedz^[(Srodek+1+Prawy) div 2]
{*}    else naPrawo:=0;
{*}    wPrzedz^[Srodek]:=Ile-1;
  if naPrawo>=r then Lewy:=Srodek+1
```

```
    else begin
     Pom:=Ile-naLewo;
     if r>Pom then begin
      Prawy:=Srodek-1;
      r:=r-Pom
     end
     else begin
      Znaleziono:=True;
      SzukajUsun:=Srodek
     end
    end {naPrawo<r}
   until Znaleziono
  end; {SzukajUsun}
  var i:Word;
begin {Permutacja}
  for i:=DlugB downto 1 do B^[i]:=SzukajUsun(B^[i]+1)
   {To przypisanie jest poprawne, poniewaz i-ty element ciagu
    B nie bedzie juz wykorzystywany.}
end; {Permutacja}
```

Algorytm zrealizowany w procedurze `SzukajUsun` jest adaptacją dla naszych potrzeb znanej metody wyszukiwania binarnego. Po każdej iteracji w pętli `repeat`, długość przedziału, w którym znajduje się poszukiwany element, jest zmniejszana co najmniej o połowę. Stąd wynika, że szukany element zostanie znaleziony po co najwyżej $\lceil \log(n+1) \rceil$ iteracjach pętli (gdyż, po co najwyżej tylu iteracjach przedział poszukiwań zostanie zawężony do przedziału jednoelementowego). Jedynym problemem, jaki jeszcze musimy rozwiązać, jest obliczanie `Ile`, `naLewo` i `naPrawo` – liczb elementów ciągu q w przedziałach, odpowiednio, `[Lewy,Prawy]`, `[Lewy,Srodek-1]` oraz `[Srodek+1,Prawy]`. W tym celu rozważmy powyższy algorytm zastosowany do ciągu $q = [n, n-1, \ldots, 1]$. Dla każdego $i = 1, \ldots, n$ oznaczmy przez *Lewy*(i) i *Prawy*(i), odpowiednio, wartości zmiennych `Lewy` i `Prawy` po zakończeniu działania algorytmu dla $k = i$. Zauważmy, że i jest środkiem przedziału [*Lewy*(i), *Prawy*(i)], tzn. $i = \lfloor (Lewy(i)+Prawy(i))/2 \rfloor$. Załóżmy, że przed l-tą iteracją pętli `for` w procedurze `Permutacja` mamy obliczone wartości w tablicy `wPrzedz[1..n]` w taki sposób, że `wPrzedz[i]` jest równe liczbie elementów ciągu q z przedziału [*Lewy*(i), *Prawy*(i)], dla każdego $i = 1, \ldots, n$.

Przykład

Niech $n = 6$, $q = [5, 2, 1]$. Wtedy mamy: 3 jest środkiem [1,5], 1 jest środkiem [1, 2], 2 jest środkiem [2, 2], 4 jest środkiem [4, 5] i 5 jest środkiem [5, 5] oraz `wPrzedz=[2,1,3,1,1]`.

Mając tak policzoną tablicę `wPrzedz` możemy łatwo obliczyć `Ile`, `naLewo` i `naPrawo` w sposób przedstawiony w instrukcjach oznaczonych na lewym marginesie przez {*} w procedurze `SzukajUsun`. Aktualizacja tablicy `wPrzedz` w trakcie wyszukiwania k-tego elementu w q pozwala bezpiecznie powtarzać algorytm dla kolejnych wartości k. Musimy jeszcze pokazać, w jaki sposób inicjować tablicę `wPrzedz` dla początkowego ciągu $q = [n, n-1, \ldots, 1]$. Wykonuje to następująca procedura rekurencyjna:

```
procedure Wypelnij_wPrzedz(Lewy,Prawy:Word);
 {Procedura rekurencyjna nadaje wartosci poczatkowe elementom
  tablicy wPrzedz^.}
 var Srodek:Integer;
begin
 if Lewy=Prawy then wPrzedz^[Lewy]:=1
 else begin
  Srodek:=(Lewy+Prawy) div 2;
  wPrzedz^[Srodek]:=Prawy-Lewy+1;
  if Lewy<Srodek then Wypelnij_wPrzedz(Lewy,Srodek-1);
  Wypelnij_wPrzedz(Srodek+1,Prawy)
 end {Lewy<>Prawy}
end; {Wypelnij_wPrzedz}
```

Po wywołaniu `Wypelnij_wPrzed(1,n)`, wartość `wPrzedz[i]` będzie równa liczbie elementów ciągu $q = [n, n-1, \ldots, 1]$ w przedziale [*Lewy*(i), *Prawy*(i)], dla każdego $i = 1, \ldots, n$. Zauważmy, że liczba operacji potrzebnych do zainicjowania tablicy `wPrzedz` jest proporcjonalna do łącznej liczby przypisań `wPrzedz[..] := ...`, wykonywanych po wywołaniu `Wypelnij_wPrzedz(1,n)`. Takich przypisań jest dokładnie n, po jednym dla każdej pozycji tablicy `wPrzedz`.

Podsumowując, wczytanie ciągu wejściowego i nadanie wartości początkowych elementom tablicy `wPrzedz` wykonuje się w czasie liniowym, zależnym od n. Następnie wykonywanych jest n iteracji algorytmu Permutacja (czyli procedury `Permutacja`). Każda iteracja trwa co najwyżej $O(\log n)$. Stąd wynika, że dla poprawnego kodu β o długości n, odpowiadającą mu permutację można wyznaczyć w czasie $O(n \log n)$.

Poniżej przedstawiamy pozostałą część programu rozwiązującego zadanie KODOWANIE PEREMUTACJI.

```
{Struktury danych zosaly zdefiniowane powyzej}
procedure Inicjalizacja (var B,wPrzedz:WskDoTablicy;
                         var DlugB:Word;
                         var PoprawnyB:Boolean);
 {Procedura wczytuje dlugosc ciagu wejsciowego B, a nastepnie
  ciag B jednoczesnie sprawdzajac, czy ciag ten jest kodem
  permutacji. Informacja o poprawnosci ciagu B jest
  przechowywana w zmiennej logicznej PoprawnyB. Jezeli ciag
  wejsciowy jeast poprawny, to budowana jest dla niego
  tablica wPrzedz.}

procedure Wypelnij_wPrzedz(Lewy,Prawy:Word);
 {Opis tej procedury zostal juz podany.}

 var f:Text;  i:Integer;
begin {Inicjalizacja}
 Assign(f,'KOD.IN');  Reset(f);  Readln(f,DlugB);
 i:=0;  New(B);  PoprawnyB:=True;
 while (i<DlugB) and PoprawnyB do begin
  i:=i+1;  Readln(f,B^[i]);
  PoprawnyB:=B^[i]<i
 end;
 Close(f);
  {Jezeli ciag B jest poprawny, to zostaje utworzona tablica
   wPrzedz^, ktora jest nastepnie wypelniana.}
 if PoprawnyB then begin
  New(wPrzedz);
  Wypelnij_wPrzedz(1,DlugB)
 end
end; {Inicjalizacja}

procedure Permutacja(B,wPrzedz:WskDoTablicy; DlugB:Word);
 {Opis tej procedury zostal juz podany.}

procedure Wyniki(B,wPrzedz:WskDoTablicy;
                 DlugB:Word; PoprawnyB:Boolean);
```



```
 var f:Text;  i:Word;
begin {Wyniki}
 Assign(f,'KOD.OUT');  Rewrite(f);
 if PoprawnyB then begin
  Permutacja(B,wPrzedz,DlugB);
  for i:=1 to DlugB do Writeln(f,B^[i])
 end
 else Writeln(f,'NIE');
 Close(f)
end; {Wyniki}

var
  n:Word;  {Dlugosc ciagu Beta}
  Beta,wPrzedziale:WskDoTablicy;
   {Beta^, wPrzedziale^ - ciag Beta i odpowiadajaca mu
    tablica, uzywana w wyszukiwaniu elementow ciagu q.}
  PoprawnyBeta:Boolean;
   {Zmienna logiczna informujaca, czy ciag Beta jest kodem
    permutacji.}
begin {Program glowny}
 Inicjalizacja(Beta,wPrzedziale,n,PoprawnyBeta);
 Wyniki(Beta,wPrzedziale,n,PoprawnyBeta)
end.
```

Omówienie rozwiązań podanych przez uczniów

Zadanie okazało się tylko pozornie łatwe. Uczniowie nie mieli kłopotów ze sprawdzeniem, czy ciąg wejściowy β jest kodem permutacji. Prawie wszyscy podali również poprawne algorytmy obliczania permutacji. Rozwiązania te były w większości oparte na ogólnej idei generowania permutacji, przedstawionej w algorytmie Permutacja. Do reprezentowania ciągu q używano zarówno list, jak i tablic. Niestety, jak wspomnieliśmy, takie rozwiązania działają w czasie proporcjanalnym do $\sum_{i=1}^{n} b_i$ – okazywały się one bardzo czasochłonne dla pewnych kodów wejściowych. Wydaje się, że większość uczniów po prostu nie dostrzegła, gdzie leży trudność w rozwiązaniu tego zadania.

Tylko kilku finalistów zauważyło, że głównym problemem w tym zadaniu jest dobór właściwej reprezentacji dla ciągu q, umożliwiającej szybką lokalizację k-tego elementu w tym ciągu. Do reprezentowania ciągu q używano binarnych drzew poszukiwań, konstruowanych w sposób jawny. W rozwiązaniu autorskim takie drzewo jest ukryte w tablicy `wPrzedz` – korzeń drzewa jest na pozycji

$\lfloor(1+n)/2\rfloor$, jego lewy syn jest na pozycji $\lfloor(1+\lfloor(1+n)/2\rfloor-1)/2\rfloor$, prawy syn znajduje się na pozycji $\lfloor(\lfloor(1+n)/2\rfloor+1+n)/2\rfloor$ itd. W jednym rozwiązaniu uczniowskim drzewo poszukiwań było również ukryte w tablicy, podobnie jak w rozwiązaniu autorskim.

Testy

Do sprawdzania działania programów uczniowskich użyto 9 testów. Cztery z nich sprawdzały poprawność rozwiązań. Długości ciągów wejściowych wynosiły co najwyżej 1000. W jednym teście ciąg nie był kodem permutacji. 5 testów sprawdzało szybkość działania programów. Ciągi wejściowe miały długości od 10000 do 30000 elementów. W 4 z tych testów suma wartości elementów ciągów wejściowych była bliska n^2. W jednym teście, ciąg wejściowy był kodem permutacji identycznościowej, tzn. (1, ..., n).

7.3.2. Zadanie OBCHODZENIE DRZEWA SKOKAMI
(Autor: Krzysztof Diks)

Treść zadania OBCHODZENIE DRZEWA SKOKAMI

Grafem nazywamy każdą parę (V, E), w której V jest skończonym zbiorem elementów zwanych **wierzchołkami grafu**, a E jest podzbiorem zbioru wszystkich nieuporządkowanych par różnych wierzchołków. Elementy zbioru E nazywamy **krawędziami** grafu. Jeżeli dla każdej pary różnych wierzchołków u, v istnieje dokładnie jeden ciąg różnych wierzchołków $w_0, w_1, ..., w_k$ taki, że $w_0 = u$, $w_k = v$ oraz pary $\{w_i, w_{i+1}\} \in E$ dla $i = 0, ..., k-1$, to graf nazywamy **drzewem**. O wierzchołkach u, v mówi się, że są odległe w drzewie o k.

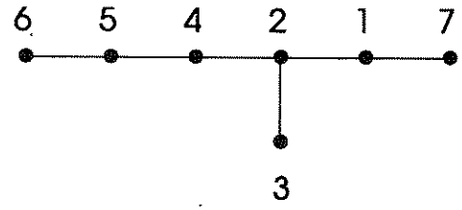
Wiadomo, że drzewo o n wierzchołkach ma dokładnie $n-1$ krawędzi. Drzewo T, którego wierzchołki są ponumerowane od 1 do n, można jednoznacznie opisać, podając liczbę wierzchołków n, a następnie odpowiedni ciąg $n-1$ par liczb naturalnych opisujący wszystkie jego krawędzie.

Dowolną permutację wierzchołków – to znaczy ciąg, w którym każdy wierzchołek drzewa występuje dokładnie jeden raz – nazywamy **porządkiem obchodzenia drzewa**. Jeżeli każde kolejne dwa wierzchołki w pewnym porządku są odległe w drzewie T co najwyżej o c, to mówimy, że jest to **porządek obchodzenia drzewa ze skokiem** c.

Wiadomo, że dla każdego drzewa można wyznaczyć porządek jego obchodzenia ze skokiem 3.

Przykład

Rysunek przedstawia drzewo o 7 wierzchołkach. Wierzchołki drzewa są reprezentowane przez czarne kropki, a krawędzie przez odcinki łączące kropki.



To drzewo można obejść ze skokiem 3 odwiedzając jego wierzchołki w następującym porządku: 7 2 3 5 6 4 1.

Zadanie

Ułóż program, który:
- wczytuje opis drzewa z pliku tekstowego DRZ.IN,
- znajduje jeden dowolny porządek obchodzenia tego drzewa ze skokiem 3,
- zapisuje wyznaczony porządek w pliku tekstowym DRZ.OUT.

Wejście

- W pierwszym wierszu pliku DRZ.IN jest zapisana liczba całkowita dodatnia n, nie większa niż 5000 – jest to liczba wierzchołków drzewa.
- W każdym z kolejnych $n-1$ wierszy jest zapisana jedna para liczb naturalnych oddzielonych pojedynczym odstępem, reprezentująca jedną krawędź drzewa.

Wyjście

W kolejnych wierszach pliku DRZ.OUT należy zapisać numery kolejnych wierzchołków w porządku obchodzenia drzewa ze skokiem trzy – każdą liczbę należy zapisać w osobnym wierszu.

Przykład

Poprawnym opisem drzewa przedstawionego na rysunku jest następujący plik DRZ.IN:

```
7
1 2
2 3
2 4
```

```
4 5
5 6
1 7
```

Poprawnym rozwiązaniem jest następujący zapis w pliku DRZ.OUT:

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

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

Rozwiązanie zadania OBCHODZENIE DRZEWA SKOKAMI

Na początku podamy kilka definicji i spostrzeżeń, które wykorzystamy w opisie naszego rozwiązania. Ciąg wierzchołków o początku w wierzchołku u i końcu w wierzchołku v, o którym jest mowa w treści zadania, nazywa się **drogą** łączącą u z v. Drzewo z wyróżnionym dokładnie jednym wierzchołkiem nazywa się **drzewem z korzeniem**, a wyróżniony wierzchołek – **korzeniem drzewa**. Niech T będzie drzewem z korzeniem w wierzchołku r. Dla każdego wierzchołka v w T, różnego od korzenia, oznaczmy przez $Poprz(v)$ wierzchołek występujący bezpośrednio po v, na jedynej drodze łączącej v z r w drzewie. Dla korzenia r przyjmuje się na ogół $Poprz(r) = 0$. Wierzchołek $Poprz(v)$ nazywamy **poprzednikiem** v w drzewie z korzeniem, a v jest wtedy **następnikiem** wierzchołka $Poprz(v)$. Następników tego samego wierzchołka nazywamy **braćmi** (a ogólniej – **dziećmi**). Poddrzewem T o korzeniu w wierzchołku v nazywamy drzewo (W, F) o korzeniu w v, w którym

$$W = \{u : v \text{ leży na drodze łączącej } u \text{ z } r \text{ w drzewie } T\}$$

a F jest zbiorem tych krawędzi z E, które łączą wierzchołki z W.

Przykład

Rozważmy drzewo z rysunku w treści zadania i wyróżnijmy w nim 4, jako korzeń. Wtedy $Poprz[1..6]$=[2, 4, 2, 0, 4, 5]. Poddrzewem z korzeniem w wierzchołku 2 jest drzewo o wierzchołkach 2, 1, 3, 7. Następnikami wierzchołka 2 są 1 i 3.

Niech T będzie drzewem. Obchodzenie drzewa T możemy rozpocząć od dowolnego wierzchołka r – od momentu wyboru wierzchołka r traktujemy T jak drzewo o korzeniu w r. Niech $w_1, w_2, \ldots, w_k$ będą następnikami r w T. Wydaje się, że najlepszym sposobem obejścia drzewa T jest obejść najpierw poddrzewo o korzeniu w w_1, potem w w_2 itd. Należy przy tym zagwarantować spełnienie następujących warunków:

- każde poddrzewo jest obchodzone ze skokiem 3,
- odległość w T między r a pierwszym odwiedzanym wierzchołkiem w poddrzewie o korzeniu w w_1 wynosi co najwyżej 3,
- odległość w T między ostatnim odwiedzonym wierzchołkiem w poddrzewie o korzeniu w w_i a pierwszym odwiedzanym wierzchołkiem w poddrzewie o korzeniu w w_{i+1}, dla każdego $i = 1, \ldots, k-1$, wynosi co najwyżej 3.

Przykład

Rozważmy 10-wierzchołkowe drzewo T o krawędziach {1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {6, 7}, {7, 8}, {8, 9} i {1, 10}. Załóżmy, że obchodzenie T rozpoczynamy od wierzchołka 1 i najpierw odwiedzamy wierzchołki z poddrzewa o korzeniu 2, a na końcu wierzchołek 10. Posuwając się w kierunku wierzchołka 9 musimy pamiętać, że po jego osiągnięciu czeka nas jeszcze powrót w kierunku korzenia 1, aby na końcu osiągnąć wierzchołek 10. W tym celu wystarczy poruszać się w kierunku wierzchołka 9 po wierzchołkach leżących w parzystych odległościach od korzenia, a wracać po wierzchołkach o odległościach nieparzystych. Otrzymywany w ten sposób porządek obchodzenia drzewa T jest następujący: 1, 3, 5, 7, 9, 8, 6, 4, 2, 10.

Naszkicowany w tym przykładzie pomysł obchodzenia drzewa ze skokiem 3 został wykorzystany w procedurze `Kolejny(u,...)`, wypisującej wierzchołki poddrzewa z korzeniem w u w porządku obchodzenia ze skokiem 3. Ponieważ procedura ta będzie wywoływana rekurencyjnie, jej dodatkowymi parametrami są: poprzednik wierzchołka u oraz informacja o tym, czy odległość u od korzenia jest parzysta.

W procedurze `Kolejny` drzewo jest reprezentowane w postaci **list sąsiedztwa**, które zostały użyte np. w rozwiązaniu zadania PRZEDSIĘWZIĘCIE (zob. *I-OI*, str. 72). W tej reprezentacji, dla każdego wierzchołka drzewa (a ogólnie grafu, gdyż w ten sposób można reprezentować dowolny graf) tworzymy listę jego sąsiadów, czyli wierzchołków połączonych z nim krawędzią. Wskaźniki do list dla poszczególnych wierzchołków umieszczamy w tablicy. Definicje odpowiednich typów danych mogą mieć następującą postać:

```
const
 Nmax=5000;
type
 NumWierzch =1..Nmax;
 Wsk        =^ElListy;
 ElListy    =record
               Wierzch:NumWierzch;
               Nast   :Wsk
             end;
 TablicaList=array[numWierzch] of Wsk;
```

Przykład

Drzewo z treści zadania ma następującą reprezentację w postaci list sąsiedztwa:

1: 2, 7
2: 1, 3, 4
3: 2
4: 2, 5
5: 4, 6
6: 5
7: 1

```
procedure Kolejny(u,v:Integer; Odl:Boolean);
  {Obchodzenie poddrzewa o korzeniu w wierzcholku u, ktorego
   poprzednikiem w drzewie jest wierzcholek v. Jezeli u jest
   korzeniem drzewa, to przyjmujemy v=0. Odl=True <=>
   odleglosc u od korzenia jest parzysta.
   f jest plikiem, do ktorego sa wpisywane wyniki.}
 var w:Wsk;
begin
 if Odl then begin
  {Jezeli u jest w odleglosci parzystej od korzenia drzewa,
   to zostaje wypisany, a nastepnie sa odwiedzane jego
   nastepniki.}
  Writeln(f,u);
  w:=Listy[u];
  while w<>Nil do begin
   if w^.Wierzch<>v then
    Kolejny(w^.Wierzch,u,False);
   w:=w^.Nast
```



```
  end
 end {u jest w odleglosci parzystej od korzenia}
 else begin
  {Wierzcholek u jest w nieparzystej odleglosci od korzenia
   drzewa. Najpierw odwiedzamy nastepniki wierzcholka u, a
   potem wypisujemy jego samego.}
  w:=Listy[u];
  while w<>Nil do begin
   if w^.Wierzch<>v then Kolejny(w^.Wierzch,u,TRUE);
   w:=w^.Nast
  end;
  Writeln(f,u)
 end {u jest w odleglosci nieparzystej od korzenia}
end; {Kolejny}
```

Aby znaleźć porządek obchodzenia całego drzewa należy wywołać tę procedurę dla dowolnego wierzchołka, jako korzenia drzewa, na przykład dla wierzchołka o numerze 1. Zatem należy wykonać `Kolejny(1,0,True)`, gdyż zgodnie z przyjętym założeniem poprzednikiem korzenia jest fikcyjny wierzchołek o numerze 0 i przyjmujemy, że odległość korzenia od siebie jest parzysta. Wtedy, jeżeli u jest wierzchołkiem w drzewie leżącym w parzystej odległości od korzenia, to po wywołaniu `Kolejny(u,...)` najpierw wypisujemy ten wierzchołek, a następnie obchodzimy poddrzewa o korzeniach będących następnikami u. Natomiast dla u leżącego w nieparzystej odległości od korzenia, najpierw obchodzimy poddrzewa, a następnie wypisujemy u.

Uzasadnienie poprawności algorytmu

Załóżmy, że drzewo T ma co najmniej 2 wierzchołki. (Dla drzewa 1-wierzchołkowego algorytm jest oczywiście poprawny.) Rozważmy wierzchołek u i zastanówmy się, jaki wierzchołek zostanie wypisany bezpośrednio po u. Mogą zajść następujące przypadki:

1. Odległość wierzchołka u od korzenia jest parzysta.

1.1. Wierzchołek u ma następnika w T.
Ponieważ odległość od korzenia każdego następnika wierzchołka u jest nieparzysta, kolejnym wierzchołkiem wypisanym po u jest następnik pewnego następnika wierzchołka u lub następnik u, który nie ma własnych następników.

1.2. Wierzchołek u nie ma następnika.
Ponieważ drzewo zawiera co najmniej 2 wierzchołki, więc u nie jest korzeniem i dlatego musi być następnikiem pewnego wierzchołka w, który znajduje się w odległości nieparzystej od korzenia.

1.2.1. Któryś z następników wierzchołka w nie został jeszcze wypisany.
Wtedy kolejnym wierzchołkiem wypisanym po u jest pewien, dotychczas niewypisany następnik wierzchołka w.

1.2.2. Wszystkie następniki wierzchołka w zostały wypisane.
Wtedy kolejnym wierzchołkiem wypisanym po u jest w.

2. Odległość wierzchołka u od korzenia jest nieparzysta.

2.1. Wierzchołek u ma niewypisanego brata.
Ponieważ każdy brat wierzchołka u leży również w odległości nieparzystej od korzenia, wierzchołkiem wypisanym bezpośrednio po u będzie następnik niewypisanego brata lub niewypisany brat, który nie ma własnych następników.

2.2. Wszyscy bracia wierzchołka u zostały już wypisane.
Niech v będzie poprzednikiem wierzchołka u w drzewie. Wtedy v leży w odległości parzystej od korzenia i został już wypisany. Jeżeli v jest korzeniem drzewa, to wszystkie wierzchołki zostały już wypisane. Załóżmy, że v nie jest korzeniem drzewa. Jeżeli v ma niewypisanego brat, to wierzchołkiem wypisanym bezpośrednio po u będzie niewypisany brat v. W przeciwnym przypadku wypisany zostanie poprzednik poprzednik wierzchołka v.

W każdym przypadku, wierzchołek wypisany bezpośrednio po u jest od niego odległy o co najwyżej 3, co dowodzi poprawności naszego algorytmu.

Złożoność algorytmu

Zauważmy, że łączna długość list sąsiedztwa n-wierzchołkowego drzewa T wynosi $2n - 2$, gdyż każde drzewo o n wierzchołkach ma $n - 1$ krawędzi i dla każdej krawędzi $\{u, v\}$, wierzchołek u występuje na liście sąsiadów v, a wierzchołek v – na liście sąsiadów u. Porządek wierzchołków na listach może być dowolny. Wszystkie listy można utworzyć znając tylko krawędzie drzewa. W tym celu wystarczy zainicjalizować listy jako puste, a następnie dla każdej krawędzi $\{u, v\}$, dołączać u do listy sąsiedztwa v, a v – do listy sąsiedztwa u. Zatem, listy sąsiedztwa dla dowolnego drzewa można zbudować w czasie proporcjonalnym do liczby krawędzi. Szczegółową realizację tworzenia listowej reprezentacji drzewa w postaci procedury w języku Pascal pozostawiamy do samodzielnego wykonania.

Niech T będzie drzewem o korzeniu r. Wtedy, wszystkie wierzchołki sąsiednie z r są jego następnikami w drzewie oraz dla każdego wierzchołka v, różnego od korzenia, dokładnie jeden jego wierzchołek sąsiedni jest jego poprzednikiem, a wszystkie inni są następnikami v. Zatem, mając listę sąsiedztwa wierzchołka v i znając jego poprzednika w drzewie (lub wiedząc, że v jest korzeniem), można zidentyfikować wszystkie następniki wierzchołka v w czasie proporcjonalnym do ich liczby. W procedurze `Kolejny`, kolejne następniki wierzchołka `u` są przeglądane w instrukcjach `while`. Czas wykonania tych instrukcji jest proporcjonalny do długości listy sąsiedztwa wierzchołka `u`, zatem złożoność czasowa wyznaczenia porządku obchodzenia całego drzewa jest proporcjonalna do sumy długości wszystkich list sąsiedztwa. Podsumowując, żądany w zadaniu porządek obchodzenia n-wierzchołkowego drzewa można wyznaczyć w czasie $O(n)$.

Omówienie rozwiązań podanych przez uczniów

Algorytmy zaproponowane przez uczniów były oparte na ogólnej idei przedstawionej w naszym rozwiązaniu, tzn. obchodzenie rozpoczynano w dowolnym wierzchołku drzewa r, a następnie obchodzone były kolejno poddrzewa z korzeniami w wierzchołkach będących następnikami r. Wśród rozwiązań uczniowskich można wyróżnić dwie metody obchodzenia poddrzew. Pierwsza metoda została opisana powyżej – przy posuwaniu się w kierunku od korzenia pomijany jest co drugi wierzchołek i pominięte wierzchołki są następnie wypisywane podczas przechodzenia w kierunku do korzenia. Metoda druga jest bardzo podobna – przy posuwaniu się w kierunku od korzenia jest wykorzystywany co trzeci wierzchołek, tzn. po wypisaniu wierzchołka v są wypisywane kolejno wierzchołki z poddrzewa o korzeniu w v odległe od v o 3, 6, ..., a wierzchołki pominięte są wypisywane w trakcie powrotu do korzenia.

W większości przypadków uczniowie reprezentowali drzewa w swoich programach w postaci list sąsiedztwa, czasem zrealizowanych za pomocą tablic dynamicznych. W niektórych rozwiązaniach, każda krawędź drzewa $\{u, v\}$ była zamieniana na dwa łuki, czyli na dwie krawędzie skierowane (u, v), (v, u). Następnie, tak otrzymane pary były porządkowane względem pierwszych składowych. Po uporządkowaniu, pary o tej samej pierwszej składowej występują kolejno, co umożliwia szybką identyfikacje sąsiadów każdego wierzchołka. Rozwiązania posługujące się taką reprezentacją wymagały jednak znacznie więcej pamięci, a dodatkową trudnością było sortowanie par wierzchołków.

Największą trudnością, jaką napotykali uczniowie przy rozwiązywaniu tego zadania, było znalezienie prostego sposobu na stwierdzanie, kiedy należy wypisać przeglądany wierzchołek – przed przejrzeniem jego poddrzew, czy po ich przeglądnięciu. Tylko w nielicznych przypadkach zaproponowana metoda była

tak prosta, jak w naszym rozwiązaniu (tj., w zależności od parzystości odległości odwiedzanego wierzchołka od korzenia). Metody bardziej skomplikowane były głównym źródłem błędów. Innym źródłem błędów było użycie iteracji zamiast rekurencji – w tym przypadku, trudno było zagwarantować, że wierzchołki są przeglądane we właściwej kolejności.

Testy

Użyto 10 testów dla sprawdzenia poprawności i efektywności rozwiązań uczniowskich. W czterech testach rozmiary danych wejściowych były niewielkie – drzewa miały od 1 do 5 wierzchołków. Głównym celem tych testów było sprawdzenie, czy programy poprawnie tworzą reprezentację drzewa dla dowolnie uporządkowanego ciągu krawędzi, a następnie – znajdują właściwy porządek obchodzenia drzewa.

Celem czterech następnych testów było sprawdzenie, czy programy uczniów działają poprawnie dla różnych rodzajów drzew, w postaci:

— drogi, czyli wszystkie wierzchołki z wyjątkiem dwóch mają dokładnie dwóch sąsiadów, a dokładnie dwa wierzchołki mają tylko po jednym sąsiedzie;
— warkocza, który składa się z sześciu 5-wierzchołkowych dróg o dokładnie jednym wspólnym końcu;
— pełnego drzewa binarnego o 1023 wierzchołkach;
— losowego drzewa o 500 wierzchołkach.

Drzewa drugiego i trzeciego rodzaju wymagają przy ich obchodzeniu skoków długości 3.

Celem dwóch ostatnich testów było sprawdzenie szybkości działania rozwiązań uczniowskich – były to: pełne drzewo binarne o 5000 wierzchołkach oraz drzewo losowe o 3000 wierzchołkach.

7.3.3. Zadanie POCHYLNIA (Autor: Andrzej Walat)

Treść zadania POCHYLNIA

Na pochylni na nabrzeżu portowym są ułożone w przypadkowej kolejności beczki w trzech kolorach: czerwone, zielone i niebieskie. Trzeba je uporządkować, za pomocą dźwigu w taki sposób, aby na początku (najniżej) były czerwone, następnie niebieskie, a na końcu (najwyżej) zielone.

Dźwig – w jednym ruchu – może podnieść do góry jednocześnie trzy sąsiednie beczki w dowolnym miejscu na pochylni i przenieść je na koniec, za beczki leżące powyżej, które stoczą się pod własnym ciężarem w dół wypełniając lukę.

Trzeba ułożyć plan pracy dźwigu dyktujący w kolejnych ruchach, którą trójkę beczek przenieść na koniec.

Ułożenie beczek na pochylni zapisujemy za pomocą ciągu l liter c, n, z, gdzie l to liczba beczek. Litery c, n, z w tym ciągu symbolizują odpowiednio beczkę czerwoną, niebieską lub zieloną. Zakładamy, że liczba beczek l jest nie większa niż 2000 oraz, że na pochylni są przynajmniej 3 beczki zielone.

Plan porządkowania beczek za pomocą dźwigu można zapisać w postaci skończonego ciągu $R = (r_1, \ldots, r_k)$ liczb całkowitych dodatnich nie większych niż l–2. Kolejny i-ty wyraz ciągu R – r_i to pozycja (licząc od dołu) najniższej z trzech beczek, jakie należy przenieść na koniec w i-tym ruchu.

Przykład

9 beczek ułożonych na pochylni w kolejności: (czerwona, zielona, niebieska, niebieska, czerwona, niebieska, zielona, zielona, niebieska) zapisujemy za pomocą ciągu: (c, z, n, n, c, n, z, z, n).

Dźwig pracujący według planu R = (6, 2, 5) będzie zmieniał ich ułożenie w następujący sposób: (c, z, n, n, c, n, n, z, z), (c, c, n, n, z, z, z, n, n), (c, c, n, n, n, n, z, z, z) i po trzech ruchach uporządkuje je w kolejności czerwone-niebieskie-zielone.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego POC.IN liczbę beczek l, a następnie ciąg l liter c, n, z określający początkowe ułożenie beczek na pochylni,
- układa plan porządkowania beczek w kolejności czerwone-niebieskie-zielone w postaci odpowiedniego skończonego ciągu liczb całkowitych dodatnich i zapisuje ten plan w pliku tekstowym POC.OUT.

Dla każdego ułożenia początkowego beczek (jeśli są przynajmniej trzy beczki zielone) istnieje wiele planów ich porządkowania w kolejności czerwone-niebieskie-zielone. Mogą się one różnić liczbą ruchów. Twój program powinien znaleźć i wypisać tylko jeden z nich. Liczba ruchów nie musi być minimalna, aby rozwiązanie było poprawne. Powinieneś jednak dążyć do tego by Twój program układał plany porządkowania beczek w możliwie małaj liczbie ruchów i aby znajdował je możliwie jak najszybciej.

Wejście

- W pierwszym wierszu pliku tekstowego POC.IN jest zapisana jedna liczba całkowita l nie mniejsza niż 3 i nie większa niż 2000. Jest to liczba beczek na pochylni.

- W każdym z kolejnych l wierszy jest zapisany jeden znak – litera c, n, albo z, określająca kolor odpowiedniej kolejnej beczki na pochylni. W tym ciągu l znaków są przynajmniej 3 litery z.

Wyjście

- W pliku tekstowym POC.OUT należy zapisać odpowiedni skończony ciąg liczb całkowitych dodatnich, który jest planem porządkowania ułożenia beczek.
- Każdą liczbę tego ciągu należy zapisać w osobnym wierszu.

Przykład

Dla pliku POC.IN:

```
9
c
z
n
n
c
n
z
z
n
```

poprawnym rozwiązaniem jest następujący ciąg w pliku POC.OUT:

```
6
2
5
```

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

Rozwiązania zadania POCHYLNIA

Podamy dwa rozwiązania tego zadania – pierwsze generuje plan porządkowania pochylni o długości $O(l^2)$, a drugie – o długości $O(l)$, gdzie l jest liczbą beczek na pochylni.

Rozwiązanie nr 1

Jeżeli na pochylni znajdują się co najmniej cztery beczki, to dowolną z nich, przenosząc po trzy beczki na górę pochylni, można przemieścić na początek. Można więc najpierw przenieść pierwszą czerwoną beczkę na początek pochylni. Następnie, przyjąć umownie, że początek pochylni znajduje się o jedną pozycję wyżej i przenieść kolejną czerwoną beczkę na (nowy) początek i tak dalej, aż na początku znajdą się wszystkie beczki czerwone. Następnie podobnie postępujemy z beczkami niebieskimi. W każdym kroku liczba beczek na pochylni, od umownego początku w górę, jest nie mniejsza niż cztery, ponieważ z założenia w treści zadania na pochylni znajdują się co najmniej trzy beczki zielone.

Opiszemy teraz algorytm przenoszenia k-tej beczki na pochylni na umowny początek *Pocz*. Algorytm ten jest zrealizowany w procedurze `Nap` zamieszczonej poniżej w procedurze `Ukladaj1`. Zakładamy, że spełnione są następujące nierówności: $Pocz \le k \le l$, gdzie l jest liczbą beczek na pochylni, oraz $l - Pocz \ge 4$.

Algorytm przemieszczenia k-tej beczki na ustalony początek

Niech $dl = l - Pocz + 1$.

Krok 1. Jeśli $dl < 6$, to zależnie od wartości k wybieramy odpowiednią sekwencję co najwyżej czterech ruchów.

Krok 2. Jeśli $dl > 5$ i liczba beczek leżących na pochylni między *Pocz* i poniżej k jest podzielna przez 3 (czyli $k - Pocz = 3n$), to n razy przenosimy trzy beczki z początku, czyli od *Pocz* na koniec.

Krok 3. W przeciwnym przypadku, najpierw przenosimy odpowiednią trójkę beczek zawierającą beczkę k na koniec pochylni w taki sposób, by po tym ruchu liczba beczek między *Pocz* a beczką znajdującą się pierwotnie na pozycji k była podzielna przez trzy. Następnie wykonujemy Krok 2, czyli przenosimy odpowiednią liczbę razy trzy beczki z (umownego) początku *Pocz* na koniec.

Jest oczywiste, że dla dowolnych dopuszczalnych wartości parametrów algorytm ten działa poprawnie.

Oznaczmy przez *lc*, *ln* i *lz* odpowiednio liczby beczek czerwonych, niebieskich i zielonych. Załóżmy, że czytając dane wejściowe, kolejność beczek na pochylni zostaje zapisana w jednowymiarowej tablicy znaków B o długości l i wyznaczone zostają liczby *lc*, *ln* i *lz*.

Powyższy algorytm porządkowania beczek jest zrealizowany w procedurze `Ukladaj1`, w której procedura `Nap` najpierw jest wywołana *lc* razy by wstawić na początek beczki czerwone, a nastepnie *ln* razy, by po czerwonych beczkach umieścić beczki niebieskie. Kolejność porządkowania beczek na pochylni jest zapisywana do pliku POC.OUT.

```
procedure Ukladaj1(l,lc,ln,lz:Integer; var B:TabZnak);
 var Pocz,k:Integer;
     f_out :Text;

 procedure Ruch(i:Integer);
  {Procedura ta wykonuje przestawienie trzech beczek,
   znajdujacych sie na pochylni od i-tego miejsca, na koniec
   pochylni.}
  var B1,B2,B3:Char;
     j        :Integer;
 begin
  Writeln(f_out,i);
  B1:=B[i];  B2:=B[i+1];  B3:=B[i+2];
  for j:=i+3 to l do B[j-3]:=B[j];
  B[l-2]:=B1;  B[l-1]:=B2;  B[l]:=B3
 end; {Ruch}

 procedure Nap(Pocz,k:Integer);
  var i,dl:Integer;
 begin
  dl:=l-Pocz+1;
  if dl>5 then begin
   if (k-Pocz) mod 3 = 0 then
    for i:=1 to (k-Pocz) div 3 do Ruch(Pocz)
   else begin
    case dl mod 3 of
     0:begin
        if k+2>l then begin
         Ruch(Pocz);  k:=k-3 end;
        Ruch(k);
        for i:=1 to dl div 3 - 1 do Ruch(Pocz)
       end; {dl podzielne przez 3}
     1:begin
        if k<Pocz+2 then begin
         Ruch(Pocz);  k:=l-1 end;
        Ruch(k-2);
        for i:=1 to dl div 3 do Ruch(Pocz)
       end; {dl dzieli sie przez 3 z reszta 1}
     2:begin
```



```
        if k=l then begin
         Ruch(Pocz);  k:=k-3 end;
        Ruch(k-1);
        for i:=1 to dl div 3 do Ruch(Pocz)
       end {dl dzieli sie przez 3 z reszta 2}
    end {case dl mod 3}
   end {k-Pocz nie dzieli sie przez 3}
  end {dl>5}
  else {dl=<5}
   if dl=5 then
    case k-Pocz of
     1:begin Ruch(Pocz);  Ruch(Pocz) end;
     2:begin Ruch(Pocz+1);  Ruch(Pocz) end;
     3:Ruch(Pocz);
     4:begin Ruch(Pocz);  Ruch(Pocz);  Ruch(Pocz) end
    end {case k-Pocz}
   else {dl<5}
    case k-Pocz of
     1:begin Ruch(Pocz); Ruch(Pocz);  Ruch(Pocz) end;
     2:begin Ruch(Pocz);  Ruch(Pocz) end;
     3:Ruch(Pocz)
    end {case k-Pocz}
 end; {Nap}

 procedure ZnakNap(Znak:Char; Pocz:Integer);
  var k:Integer;
 begin
  k:=Pocz;
  while B[k]<>Znak do Inc(k);
  Nap(Pocz,k)
 end; {ZnakNap}

begin {Ukladaj}
 Assign(f_out,'POC.OUT');  Rewrite(f_out);
 for Pocz:=1 to lc do ZnakNap('c',Pocz);
 for Pocz:=lc+1 to lc+ln do ZnakNap('n',Pocz);
 Close(f_out)
end; {Ukladaj1}
```

Ocena złożoności rozwiązania nr 1

Dla oceny złożoności podanego rozwiązania określimy:

- zależność długości planu porządkowania beczek od liczby beczek,
- zależność czasu działania algorytmu od liczby beczek.

Liczba ruchów, jakie trzeba wykonać, aby przemieścić k-tą beczkę na początek nieuporządkowanego obszaru pochylni, zależy od jej położenia i długości porządkowanego obszaru. Na ogół (średnio w 2/3 przypadków), ta liczba jest równa dl div 3 + 1 lub dl div 3 + 2, gdzie dl jest długością porządkowanego obszaru. Jeśli $(k - Pocz)$ mod 3 = 0, to może być nawet mniejsza – średnio w tych przypadkach wynosi dl div 6. Po przeniesieniu kolejnej beczki na początek, długość nieuporządkowanego obszaru maleje. W najgorszym przypadku długość planu porządkowania pochylni nie powinna być większa niż:

$$l/3 * l/2 = l^2/6.$$

Zatem podany algorytm znajduje plan porządkowania beczek o długości proporcjonalnej do kwadratu liczby beczek na pochylni.

Symulacja jednego ruchu ramienia dźwigu za pomocą przemieszczenia trzech kolejnych znaków w jednowymiarowej tablicy B od pozycji k na koniec tablicy wiąże się z przemieszczeniem o trzy miejsca do przodu wszystkich znaków zajmujących w tablicy B pozycje od $k + 3$ do końca. Ponieważ opisany algorytm generuje plan porządkowania pochylni, którego długość jest proporcjonalna do l^2, więc czas wykonania algorytmu jest proporcjonalny do sześcianu liczby beczek l^3.

Rozwiązanie nr 2

Podamy teraz lepsze rozwiązanie tego zadania, które generuje plan porządkowania pochylni o długości proporcjonalnej do liczby beczek l.

Podstawowym krokiem tego rozwiązania jest procedura `NaPoczatek`, która symuluje przeniesienie wszystkich lk beczek wybranego koloru *Kolor*, znajdujących się na pochylni nie niżej niż ustalone miejsce *Początek*, na początek tego obszaru. Jednocześnie zapisuje każdy ruch do pliku tekstowego POC.OUT, tworząc odpowiedni fragment planu porządkowania pochylni.

Algorytm przenoszenia beczek ustalonego koloru na początek

Rozróżniamy dwa rodzaje beczek: beczki wybranego koloru oraz inne.

Krok 1. Jeśli liczba beczek wybranego koloru lk (położonych nie niżej niż *Początek*) jest mniejsza niż pięć, to przenosimy je po kolei, jak w rozwiązaniu nr 1.

Krok 2. W przeciwnym przypadku, dążymy do utworzenia na pochylni obszaru, w którym inne beczki będą występowały trójkami. Chcemy, aby w tym obszarze

znalazły się wszystkie beczki innego koloru, z wyjątkiem tylko jednej albo dwóch, jeżeli ich liczba nie jest podzielna przez trzy. W takim przypadku powinny się one znaleźć w pierwszej trójce od miejsca *Początek*, tworząc układ ddx albo dxx, gdzie d oznacza ustalony kolor, a x – dowolny inny kolor.

Krok 3. Przenosimy pierwszą trójkę (która ma postać dxx lub ddx) na koniec, a następnie przenosimy na koniec kolejne trójki innego koloru xxx. W ten sposób otrzymujemy ułożenie, w którym na początku występują wszystkie beczki danego koloru, a powyżej – inne.

W tym algorytmie musimy wyjaśnić sposób realizacji kroku 2.

- Jeśli liczba beczek innego koloru li nie jest podzielna przez trzy, to początkową trójkę można łatwo utworzyć przez przeniesienie na początek jednej lub dwóch beczek danego koloru, a następnie odpowiednio dwóch albo jednej beczki innego koloru. Stosujemy w tym celu algorytm zrealizowany w procedurze `Nap` (jej treść znajduje się w procedurze `Ukladaj1` w rozwiązaniu nr 1).
- Następnie nad tą trójką umieszczamy dodatkowe dwie beczki wybranego koloru i otrzymujemy piątkę ddxdd lub dxxdd. W ten sposób tworzymy strefę ochronną w postaci dwóch beczek dd, które zabezpieczają początkową trójkę.
- Ustalamy granice obszaru, w którym beczki innych kolorów występują trójkami na końcu pochylni, powyżej ostatniej beczki. Robimy to nadając wartość $l + 1$ zmiennym *Głowa* i *Ogon*. Na początku jest to obszar pusty – odnotowujemy, że liczba beczek innych kolorów, leżących od miejsca *Ogon* w górę wynosi $lio = 0$.
- Następnie krok po kroku poszerzamy ten obszar zmieniając jego granice *Głowa* i *Ogon*. Szukamy beczek innych kolorów w obszarze poniżej miejsca *Głowa* i przenosimy je na koniec, zmieniając odpowiednio wartości zmiennych lio i *Głowa*. Jeśli liczba beczek innych kolorów za miejscem *Ogon* spełnia $lio \geq 3$, to porządkujemy strefę od miejsca *Ogon* w górę przenosząc kolejno trzy beczki innych kolorów na jej początek, a następnie odpowiednio przenosimy miejsce *Ogon* o trzy miejsca w górę. Po wykonaniu tej operacji staramy się skrócić obszar od miejsca *Ogon* w górę. Jeśli $lio = 0$, to przenosimy *Ogon* na sam koniec $(l + 1)$. W przeciwnym przypadku (tj. gdy $lio = 1$) przenosimy *Ogon* do góry na pozycję zajmowaną przez jedyną w tym obszarze beczkę innego koloru. Jeżeli powyżej niej jest odpowiednio dużo beczek wybranego koloru, to przenosimy trójkę xdd na koniec, a *Ogon* – na pozycję $l - 2$. W ten sposób obszar, w którym beczki innego koloru występują trójkami, poszerza się. Jednocześnie strefa od miejsca *Ogon* w górę, w której wykonujemy większość ruchów, nadmiernie nie wydłuża się.

W treści procedury Ukladaj2, która tworzy plan porządkowania pochylni, występują dwa wywołania procedury NaPoczatek: najpierw jest symulowane przeniesienie wszystkich beczek czerwonych na dół pochylni i jednocześnie powstaje plan wykonania tego zadania, a następnie są przenoszone wszystkie beczki niebieskie na początek obszaru powyżej czerwonych beczek, czyli od pozycji $lc + 1$.

```
procedure Ukladaj2(l,lc,ln,lz:Integer; var B:TabZnak);
 var Pocz,k,Glowa,Ogon,LWTr,ltr:Integer;

 {W tym miejscu nalezy umiescic opisy procedur Ruch, Nap,
  i ZnakNap znajdujace sie w tresci procedury Ukladaj1.}

 procedure NieZnakNap(Znak:Char; Pocz:Integer);
  var k:Integer;
 begin
  k:=Pocz;
  while B[k]=Znak do Inc(k);
  Nap(Pocz,k)
 end; {NieZnakNap}

 procedure NaPoczatek(Kolor:Char; Poczatek,lk:Integer);
  var li,lio,Pocz,k:Integer;
  procedure Przygotowanie;
  begin
   li:=l+1-Poczatek-lk;
   if li mod 3 > 0 then begin
    ZnakNap(Kolor,Poczatek);
    case li mod 3 of
     1:ZnakNap(Kolor,Poczatek+1);
     2:NieZnakNap(Kolor,Poczatek+1)
    end;
    NieZnakNap(Kolor,Poczatek+2);
    ZnakNap(Kolor,Poczatek+3);  ZnakNap(Kolor,Poczatek+4)
   end {li nie jest podzielne przez 3}
   else begin
    ZnakNap(Kolor,Poczatek);  ZnakNap(Kolor,Poczatek+1)
   end;
   Glowa:=l+1;  Ogon:=l+1;
```



```
   lwtr:=li div 3;  ltr:=0;  lio:=0
  end; {Przygotowanie}
  procedure PrzeniesInnedoOgona;
   var i,LiczI:Integer;
  begin
   repeat Dec(Glowa) until B[Glowa]<>Kolor;
   LiczI:=1;
   Dec(Glowa);  if B[Glowa]<>Kolor then Inc(LiczI);
   Dec(Glowa);  if B[Glowa]<>Kolor then Inc(LiczI);
   if LiczI<3 then begin
    Ruch(Glowa);  Ogon:=Ogon-3;  lio:=lio+LiczI
   end
   else Inc(ltr)
  end; {PrzeniesInnedoOgona}
  procedure PorzadkujOgon;
   var Pocz:Integer;
  begin
   for Pocz:=Ogon to Ogon+2 do NieZnakNap(Kolor,Pocz);
   Ogon:=Ogon+3;
   Inc(ltr);  lio:=lio-3;
   if lio=0 then Ogon:=l+1
   else begin
    while B[Ogon]=Kolor do Inc(Ogon);
    if Ogon<l-2 then begin
     Ruch(Ogon);  Ogon:=l-2
    end
    else Ogon:=l-2
   end {lio<>0}
  end; {PorzadkujOgon}

begin {NaPoczatek}
 if lk<5 then
  for Pocz:=Poczatek to Poczatek+lk-1 do ZnakNap(Kolor,Pocz)
 else begin
  Przygotowanie;
  while ltr<lwtr do
   if lio<3 then PrzeniesInnedoOgona else PorzadkujOgon;
  if li mod 3 > 0 then Ruch(Poczatek);
  Pocz:=Poczatek;
```

```
    for k:=1 to lwtr do begin
     while B[Pocz]=Kolor do Inc(Pocz);
     Ruch(Pocz)
    end
   end {lk>=5}
 end; {NaPoczatek}

begin {Ukladaj2}
 Assign(f_out,'POC.OUT'); Rewrite(f_out);
 NaPoczatek('c',1,lc);
 NaPoczatek('n',lc+1,ln);
 Close(f_out)
end; {Ukladaj2}
```

Ocena złożoności rozwiązania nr 2

Oceńmy najpierw z ilu ruchów może się składać plan przenoszenia wszystkich czerwonych beczek na dół pochylni, utworzony przez wywołanie procedury: `NaPoczatek('c',1,lc)`.

Jeśli liczba czerwonych beczek spełnia $lc < 5$, to liczba ruchów, jakie trzeba wykonać, by je przenieść na początek, jest nie większa niż $4l/3$.

Jeżeli $lc > 4$, to:

- liczba ruchów przygotowawczych, związanych z utworzeniem na początku pochylni sekwencji postaci ccxcc lub cxxcc, jest nie większa niż $5l/3$;
- liczba ruchów związanych z przenoszeniem beczek innego koloru niż czerwony z obszaru poniżej miejsca *Głowa* poza miejsce *Ogon*, nie może być większa niż $l/3$;
- jedno uporządkowanie obszaru za miejscem *Ogon*, tj. przeniesienie trzech beczek innego koloru na początek tego obszaru, daje się wykonać w liczbie ruchów nie większej niż $3 \cdot 4 = 12$ (bo liczba beczek w tym obszarze nie może być większa niż 9); wykonujemy je nie więcej niż $l/3$ razy; w sumie na wszystkie porządkowania poza miejscem *Ogon* potrzeba nie więcej niż $4l$ ruchów;
- jedno skrócenie obszaru poza miejscem *Ogon* wymaga jednego ruchu, w sumie $l/3$ ruchów;
- przeniesienie beczek innego koloru trójkami na koniec można wykonać w co najwyżej $l/3$ ruchach.

W sumie, przemieszczenie wszystkich czerwonych beczek na początek można wykonać w liczbie ruchów nie większej niż $5l$. Również tyle co najwyżej ruchów potrzeba, by następnie przemieścić wszystkie niebieskie beczki na początek obszaru powyżej beczek czerwonych. Zatem procedura `Układaj2` tworzy plan porządkowania beczek na pochylni w liczbie ruchów nie większej niż $10l$.

W praktyce liczba ruchów jest średnio znacznie mniejsza, niż wynika to z podanych oszacowań (zob. Tabela 1).

Każdy ruch dźwigu jest symulowany za pomocą przeniesienia trzech beczek na koniec pochylni, czemu odpowiada w realizacji algorytmu przeniesienie trzech znaków na koniec tablicy i przesunięcie średnio połowy elementów tablicy o trzy pozycje do przodu. Zatem czas działania programu jest $O(l^2)$.

Jest to konsekwencją wyboru w implementacji algorytmu tablicy do pamiętania kolorów kolejnych beczek na pochylni – przemieszczeniu trzech beczek na koniec pochylni odpowieda przesunięcie w tablicy o trzy miejsca do przodu kolorów beczek znajdujacych się powyżej przemieszczanych beczek. Ten sam algorytm można zrealizować reprezentując ułożenie beczek na pochylni za pomocą dynamicznej struktury danych (listy). Wtedy czas przeniesienia trzech beczek na koniec pochylni będzie stały i czas wykonania całego programu będzie proporcjonalny do długości generowanego planu, a więc będzie zależny liniowo od liczby beczek l.

Autorowi rozwiązania wydawało się, że wytłumaczenie algorytmu generowania planu porządkowania pochylni będzie trochę bardziej czytelne dla układu beczek reprezentowanego za pomocą tablicy.

Omówienie rozwiązań podanych przez uczniów

Spośród sześciu rozwiązań tego zadania podanych przez uczniów, które w finale uzyskały ocenę ponad 80 punktów, jeden program był realizacją algorytmu opisanego w rozwiązaniu nr 1 i generował plany porządkowania pochylni mające dokładnie taką samą długość, jak odpowiednie plany generowane przez program będący realizacja naszego rozwiązania.

Czterech zawodników zastosowało ulepszone wersje algorytmu, umożliwiające skrócenie procesu porządkowania pochylni opierającego się, jak w rozwiązaniu nr 1, na przemieszczaniu beczek po jednej na początek porządkowanego obszaru. Ulepszone programy generowały plany dwu/trzykrotnie krótsze niż nasz program.

Jeden zawodnik zastosował podobny algorytm jak w rozwiązaniu nr 2. Jego główna idea polegała na grupowaniu trójkami beczek, które ostatecznie mają się znaleźć na końcu pochylni, a następnie przeniesieniu tych trójek jedna po drugiej na koniec. Niestety, sporadycznie ten program zapętlał się dla pewnych danych, a w przypadkach, gdy dawał wynik, długość generowanego planu była zbliżona do długości planu generowanego przez nasz program będący realizacją rozwiązania nr 2.

Tabela 1.

l	Rozwiązanie nr 1	Zawodnik nr 29	Zawodnik nr 2	Rozwiązanie nr 2
20	26	29	21	25
50	253	245	91	136
100	580	690	191	308
200	3028	1560	zapętlenie	472
300	6368	3111	1008	650
400	11656	6113	1346	867
500	18673	6801	1081	1230
600	27984	7700	2239	1506
700	36887	13886	3280	2082
800	48529	18716	2622	1399
900	57064	17358	3276	1448
1000	76264	29488	2927	2665
1200	114173	35969	6399	2154

Dla porównania załączamy w Tablicy 1 długości planów porządkowania pochylni generowane przez programy: z rozwiązania nr 1, zawodników nr 29 i nr 2 oraz z rozwiazania nr 2, dla różnych losowych układów beczek na pochylni o długościach l od 20 do 1200.

Ten eksperyment potwierdza ustalone teoretycznie zależności długości tworzonych planów od liczby beczek na pochylni. W rozwiązaniu nr 1 jest to zależność kwadratowa, a w drugim – liniowa. Jednocześnie widać, że ustalony eksperymentalnie współczynnik zależności liniowej (dla losowych układów beczek) jest nieznacznie większy niż 2.

Testy

Rozwiązania uczniowskie były oceniane na podstawie wyników dziesięciu testów, które można podzielić na dwie grupy. W pierwszych pięciu testach liczba beczek była bardzo mała, od 4 do 9. Tworzyły one następujące układy: cznncnzzn, zzzc, zzzcczc, zzzcczcc, zzzzzzzzc. W pięciu następnych testach liczba beczek była coraz większa: 50, 99, 250, 750 i 2000, i na ich podstawie można było porównać efektywność rozwiązań i ocenić zależność długości generowanego planu i czasu działania programu od liczby beczek na pochylni.

7.3.4. Zadanie SKŁADANIE SŁÓW (Autor: Wojciech Rytter)

Treść zadania SKŁADANIE SŁÓW

Dane jest słowo w stanowiące wzorzec oraz skończony ciąg niepustych słów $C = (w_1, \ldots, w_k)$. W ciągu C trzeba wskazać słowa, które po złączeniu, w takiej kolejności, w jakiej występują w ciągu C, utworzą wzorzec. Rozwiązaniem ma być rosnący ciąg numerów słów danego ciągu C, które po złączeniu utworzą wzorzec. Wzorzec w i każde słowo w ciągu C składają się z co najwyżej 150 małych liter alfabetu angielskiego od 'a' do 'z' i nie zawierają znaków narodowych. Liczba słów w ciągu jest dodatnia i nie większa niż 200.

Przykład

Wzorzec rytter można utworzyć ze słów ciągu C = (ry, r, yt, y, tt, t, e, te, r, er) wybierając kolejno i łącząc słowa o numerach (2, 4, 5, 7, 9). Wybranie słów (1, 5, 10) jest innym sposobem otrzymania tego samego wzorca.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego SLO.IN następujące dane: wzorzec w, liczbę słów ciągu C, a następnie kolejne słowa tego ciągu,
- jeżeli nie istnieje rozwiązanie zadania utworzenia wzorca w z słów ciągu C, zgodnie z podanymi wyżej warunkami, to zapisuje w pliku SLO.OUT jedno słowo NIE,
- jeżeli istnieją rozwiązania i jest ich mniej niż 1000000, to zapisuje w pliku tekstowym SLO.OUT liczbę rozwiązań, a następnie jedno dowolne rozwiązanie zadania,
- jeżeli rozwiązań jest więcej niż 999999, to zapisuje w pliku SLO.OUT liczbę 1000000, a nastepnie jedno dowolne rozwiązanie zadania.

Wejście

- W pierwszym wierszu pliku SLO.IN jest zapisane jedno słowo złożone z co najwyżej 150 małych liter alfabetu angielskiego. Jest to wzorzec w.
- W drugim wierszu jest zapisana dodatnia liczba całkowita $k \leq 200$. Jest to liczba słów ciągu C.
- W kolejnych k wierszach są zapisane kolejne niepuste słowa tworzące ciąg C, każde słowo jest zapisane w osobnym wierszu i składa się z co najwyżej 150 małych liter alfabetu angielskiego. Pierwsza litera słowa jest zawsze pierwszym znakiem w wierszu, a bezpośrednio po ostatniej jest koniec wiersza.

Wyjście

W pliku SLO.OUT należy zapisać:

- albo jedno słowo NIE, w przypadku gdy ze słów w ciągu C nie można ułożyć wzorca w sposób zgodny z warunkami zadania,
- albo w pierwszym wierszu jedną liczbę, to jest liczbę rozwiązań zadania, albo 1000000, a w kolejnych wierszach rosnący ciąg numerów słów wybranych z ciągu C, które po złożeniu utworzą wzorzec – każdą liczbę w osobnym wierszu.

Przykład

Dla pliku SLO.IN:

```
rytter
10
ry
r
yt
y
tt
t
e
te
r
er
```

w pliku SLO.OUT można zapisać:

```
9
2
4
5
7
9
```

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

Rozwiązanie zadania SKŁADANIE SŁÓW

Rozwiązanie tego zadania sprowadzimy do zadania istnienia i wyznaczenia liczby dróg w odpowiednio zdefiniowanym grafie.

Niech $w = a_1 a_2 \ldots a_n$ oznacza wzorzec, a $C = \{w_1, w_2, \ldots, w_k\}$ – zbiór słów. Zdefiniujmy następująca tablicę o wymiarach $n \times k$ i wartościach logicznych:

$$PASUJE\ [i,j] := (w[i+1\ ..\ i+|w_j|] = w_j),$$

gdzie $|w_j|$ oznacza długość słowa w_j, a $w[k..l]$ oznacza fragment wzorca złożony z liter od k-tej do l-tej. Mamy $PASUJE\ [i,j] = True$, jeśli słowo w_j występuje w słowie w na pozycji i+1, a dokładniej – wystąpienie w_j w słowie w zaczyna się na pozycji i+1.

Skonstruujmy następujacy graf skierowany acykliczny $G = (V,U)$, w którym zbiór wierzchołków V odpowiada parom liczb $V = \{ (i,j) : 0 \le i \le n,\ 0 \le j \le k \}$. Dodatkowo dołączmy do V specjalny wierzchołek $u = (n+1, n+1)$ zwany **ujściem**. Wierzchołek (0,0) nazywamy **wejściem** grafu G. U jest zbiorem łuków (czyli skierowanych krawędzi), które definiujemy następująco:

$(i,j) \to (i + |w_t|, t)$ jest łukiem, gdy $PASUJE\ [i,t]$ oraz $j < t$.}

Ponadto, prowadzimy łuk od każdego wierzchołka (n,j) do ujścia.

Łatwo zauważyć, że prawdziwy jest następujący fakt:

Każdej drodze w grafie G od wejścia do ujścia odpowiada dokładnie jedna poprawna dekompozycja wzorca w na słowa ze zbioru C, zatem liczba takich dekompozycji jest równa liczbie dróg od wejścia do ujścia w grafie G.

Algorytm rozwiązujący zadanie może się więc składać z następujących kroków:

Krok 1. Oblicz wartości elementów w tablicy *PASUJE*.

Krok 2. Skonstruuj graf G.

Krok 3. Wyznacz liczbę dróg od wejścia do ujścia w grafie G.

Krok 4. Wypisz dekompozycję odpowiadającą jednej z dróg (jeśli istnieje jakakolwiek droga od wejścia do ujścia w G).

Opiszemy teraz realizację poszczególnych kroków algorytmu.

Krok 1.

Wartości elementów w tablicy *PASUJE* można obliczyć bezpośrednio z ich definicji, dla każdej pary i,j osobno. Jednakże można zastosować bardziej efektywną metodę: obliczyć dla ustalonego j wartości $PASUJE\ [i,j]$ dla wszystkich i, korzystając z szybkiego algorytmu dla problemu **dopasowywania wzorca**. Problem ten polega na znalezieniu w danym tekście wszystkich wystąpień danego słowa zwanego **wzorcem**. W naszym przypadku tekstem jest w, a wzorcem jest w_j. Jeżeli w_j występuje w słowie w począwszy od pozycji i+1 to $PASUJE\ [i,j] = True$, w przeciwnym razie $PASUJE\ [i,j] = False$.

Istnieje wiele algorytmów dla problemu dopasowania wzorca, działających w czasie liniowym. Najbardziej znanym jest algorytm Knutha-Morrisa-Pratta. Jego opis można znaleźć w książce: L. Banachowski, A. Kreczmar, W. Rytter, *Analiza algorytmów i struktur danych*, WNT, Warszawa 1989. Korzystając z tego algorytmu, tablicę *PASUJE* można wypełnić w czasie $O(nm)$, gdzie $n = |w|$ i $m = \sum_{j=1}^{k} |w_j|$.

Krok 2.

Graf G jest acykliczny, tzn. nie zawiera skierowanego cyklu, czyli zamkniętego ciągu łuków, ustawionych zgodnie z ich orientacją w grafie. Większość algorytmów stosowanych do grafu acyklicznego zyskuje na efektywności, jeśli jego wierzchołki są ponumerowane w kolejności topologicznej, czyli takiej, dla której jakikolwiek łuk w grafie prowadzi zawsze od wierzchołka o mniejszym numerze do wierzchołka o większym numerze – szczegółowe rozważania na ten temat i odpowiednie algorytmy są zamieszczone w *I-OI*, str. 72–76. Zauważmy, że w przypadku grafu z tego zadania, wierzchołkom odpowiadają pary liczb i jeśli dwie takie pary są połączone łukiem, to oba elementy w tej drugiej parze są większe od odpowiednich elementów w pierwszej parze. Zatem, porządek topologiczny wierzchołków w tym grafie może być określony przez jakiekolwiek leksykograficzne (takie, jak w słowniku) uporządkowanie par liczb odpowiadających wierzchołkom.

Graf G można reprezentować w komputerze w postaci macierzy sąsiedztwa wierzchołków lub w postaci listowej. Ta pierwsza reprezentacja zajmuje jednak zbyt dużo miejsca w pamięci. Po pierwsze, jeśli wierzchołki grafu są ponumerowane topologicznie, to dolna trójkątna część tej macierzy będzie wypełniona samymi elementami *False*. Po drugie zaś, graf w naszym zadaniu jest z reguły rzadki, tzn. ma mało łuków w stosunku do kwadratu liczby wierzchołków (czyli w stosunku do rozmiaru macierzy sąsiedztwa). W takiej sytuacji, znacznie oszczędniejszą reprezentacją naszego grafu jest reprezentacja listowa, w której dla każdego wierzchołka tworzymy listę wierzchołków, do których prowadzi łuk z tego wierzchołka. Tę reprezentację grafu możemy otrzymać bezpośrednio z tablicy *PASUJE*.

Krok 3. (Główna część algorytmu)

Aby wyznaczyć liczbę dróg od wejścia do ujścia w grafie G, obliczamy wartości elementów w tablicy *ILE*, w której *ILE* $[v]$ jest liczbą dróg z wierzchołka v do ujścia. Załóżmy, że *ILE* [*ujście*] = 1. Następne elementy tablicy *ILE* wyznaczamy w kolejności odwrotnej do uporządkowania topologicznego znalezionego w poprzednim kroku. Załóżmy, że z wierzchołka v prowadzą w grafie G łuki do wierzchołków $v_1, v_2, \ldots, v_r$. Wtedy:

$$ILE\,[v] = \sum_{i=1}^{r} ILE\,[v_i].$$

Obliczenie wszystkich elementów w tablicy *ILE* wykonujemy w czasie liniowym ze względu na rozmiar grafu G (tzn. liczbę wierzchołków plus liczbę łuków). Liczba wszystkich dekompozycji tekstu w na słowa ze zbioru C jest więc równa *ILE* [*wejście*].

Krok 4.

Dysponując tablicą *ILE* możemy łatwo wypisać jedną z dekompozycji słowa w, jeśli tylko istnieje, której w grafie G odpowiada droga od wejścia do ujścia. W tym celu przechodzimy w grafie G od wejścia do ujścia wybierając w wierzchołku v, w którym jesteśmy, wierzchołek u, do którego istnieje łuk z v oraz *ILE* $[u] \neq 0$.

Omówienie rozwiązań podanych przez uczniów

Większość zawodników nie formułowała rozwiązań w terminach teorii grafów, tym niemniej rozumowano w sposób podobny do podanego powyżej. Wartości elementów w tablicy *ILE*, będące w naszym rozwiązaniu liczbą dróg w grafie, odpowiadają liczbom dekompozycji sufiksów (czyli końcowych części) tekstu w na słowa ze zbioru C. Język teorii grafów jest wygodny w opisie algorytmu na wyższym poziomie, w programie graf nie musi występować jawnie.

Obliczenia w krokach 3 i 4 naszego algorytmu są odmianą **metody programowania dynamicznego**.

Niektórzy z uczniów zauważyli, że w rozwiązaniu można zastosować algorytm Knutha-Morrisa-Pratta, jednak w przypadku, gdy słowa ze zbioru C są stosunkowo krótkie, użycie tego algorytmu nie poprawia specjalnie efektywność rozwiązania.

Testy

Jedną grupę testów stanowiły zestawy danych, które miały sprawdzać przede wszystkim poprawność działania programów uczniowskich. Wśród nich, pierwszym testem był przykład z treści zadania, a innymi były następujące zestawy (dla oszczędności miejsca każdy zestaw piszemy w jednym wierszu):

```
a 1 a
kaczka 2 kaczka kaczka
eldorado 3 el do ra
koparka 6 a ko p r a ka
kapitan 3 kap pit tan
```

W następnym teście wzorzec miał maksymalną liczbę 150 liter, a słowa w zbiorze C miały około lub ponad 100 liter.

Wśród testów nie mogło zabraknąć jako wzorca jednego z najdłuższych słów w języku polskim:

konstantynopolitanczykowianeczka.

Zadaniem dwóch ostatnich testów było sprawdzanie efektywności rozwiązań uczniowskich. W jednym – wzorcem było słowo

abababababababcababababababab

a zbiór C tworzyło 40 słów ab, których ciąg w połowie był przedzielony słowem bca. Jak łatwo zauważyć, zadanie nie ma rozwiązania dla tych danych, chociaż na wiele sposobów można utworzyć ze słów zbioru C początkowy i końcowy fragment wzorca złożony z sześciu słów ab.

W ostatnim teście wzorzec składał się ze 150 liter a i zbiór C zawierał 200 słów będących pojedynczymi literami a. Dla tych danych zadanie ma ponad 999999 rozwiązań.

7.3.5. Zadanie SZEREGOWANIE CZYNNOŚCI (Autor: Marcin Jurdziński)

Treść zadania SZEREGOWANIE CZYNNOŚCI

Danych jest n niezależnych i niepodzielnych czynności, ponumerowanych od 1 do n. Należy je wykonać sekwencyjnie w dowolnej kolejności. Wykonanie każdej czynności trwa tym dłużej im później ją rozpoczniemy – ściśle czas wykonania czynności i wynosi $h_i(t) = a_i t + b_i$, jeśli rozpoczniemy ją w chwili t. Zakładamy, że $0 \le a_i \le 1$, $0 \le b_i \le 1$.

Należy uszeregować czynności w takiej kolejności, aby łączny czas ich wykonania był najmniejszy.

Zadanie

Napisz program, który:

- wczytuje z pliku tekstowego SZE.IN liczbę czynności n nie większą niż 10000 oraz kolejno dla każdej czynności i – współczynniki a_i oraz b_i określające zależność czasu wykonania tej czynności od chwili jej rozpoczęcia,
- znajduje takie uszeregowanie czynności, żeby łączny czas ich wykonania był minimalny i zapisuje w pliku tekstowym SZE.OUT numery czynności w takiej kolejności, w jakiej należy je wykonywać.

Wejście

- W pierwszym wierszu pliku SZE.IN jest zapisana jedna liczba całkowita dodatnia nie większa niż 10000. Jest to liczba czynności n.
- W każdym z n kolejnych wierszy jest zapisana para liczb rzeczywistych nieujemnych w standardowej notacji z kropką i sześcioma cyframi po kropce. Są one oddzielone pojedynczym odstępem. Jest to para współczynników a_i oraz b_i określających zależność czasu wykonania odpowiedniej i-tej czynności od chwili jej rozpoczęcia.

Wyjście

W pliku SZE.OUT należy zapisać uszeregowanie czynności, to znaczy odpowiednią permutację liczb 1, ..., n; każdą liczbę w osobnym wierszu.

Przykład

Dla pliku SZE.IN

```
5
0.002000 0.003000
0.016000 0.001000
0.100000 0.300000
0.016000 0.005000
0.030000 0.060000
```

w pliku SZE.OUT należy zapisać:

```
2
4
1
5
3
```

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

Rozwiązanie zadania SZEREGOWANIE CZYNNOŚCI

Uwaga. W powyższym sformułowaniu zadania nie ma żadnego założenia co do chwili rozpoczęcia wykonywania czynności. Jak wykaże analiza problemu, założenie takie nie jest istotne dla rozwiązania zadania. Jednak ze względu na pytania i wątpliwości uczestników finału Olimpiady, w trakcie zawodów przyję-

to i podano do wiadomości wszystkich uczestników upraszczające założenie, że wykonywanie czynności rozpoczyna się w chwili $t_0 = 0$.

Naszym celem jest znalezienie takiej permutacji liczb 1, ..., n, aby łączny czas wykonania wszystkich czynności był najmniejszy. Dla dowolnego ustalonego uszeregowania czynności $d_1, d_2, \ldots, d_n$, jeśli wykonywanie czynności rozpoczyna się w chwili t_0, to koszt wykonania pierwszej z nich (oznaczonej numerem d_1) wynosi:

$$t_{d_1} = h_{d_1}(t_0) = a_{d_1} t_0 + b_{d_1}.$$

Ponieważ zależy nam na tym, by wszystkie czynności wykonać jak najszybciej, możemy – nie naruszając wymagania aby czynności były niepodzielne – rozpocząć wykonywanie drugiej z kolei czynności już w chwili zakończenia wykonywania pierwszej, trzeciej z nich w chwili zakończenia drugiej itd. Jeśli zatem oznaczymy przez t_{d_i} czas wykonywania i-tej czynności w uszeregowaniu $d_1, d_2, \ldots, d_n$, to zachodzą następujące związki:

$$t_{d_2} = h_{d_2}(t_{d_1}),$$
$$t_{d_3} = h_{d_3}(t_{d_1} + t_{d_2}),$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$t_{d_n} = h_{d_n}\left(\sum_{i=1}^{n-1} t_{d_i}\right).$$

Wtedy łączny czas wykonania wszystkich czynności dla uszeregowania $d_1, d_2, \ldots, d_n$ wynosi:

$$T(d_1, d_2, \ldots, d_n) = \sum_{i=1}^{n} t_{d_i}$$

Obliczenie tych wielkości dla ustalonej permutacji $d_1, d_2, \ldots, d_n$ nie przedstawia specjalnej trudności. Rozwiązanie polegające na obliczeniu łącznego czasu wykonania czynności dla wszystkich możliwych permutacji i wybraniu takiej, dla której czas ten jest najmniejszy, nie wchodzi jednak w rachubę, ponieważ liczba permutacji ciągu n-elementowego wynosi $n!$, zatem rośnie bardzo szybko wraz ze wzrostem n.

Okazuje się, że aby znaleźć optymalne uszeregowanie, nie tylko nie potrzeba rozważać wszystkich możliwych permutacji, ale co więcej nie jest konieczne obliczanie czasów wykonywania czynności dla żadnego uszeregowania! Aby się o tym przekonać, zbadajmy kiedy zamiana kolejności dwóch sąsiadujących ze sobą w pewnym uszeregowaniu czynności może się opłacać, tzn. prowadzi do

zmniejszenia łącznego czasu ich wykonywania. Niech w uszeregowaniu $d_1, d_2, \ldots, d_n$ będzie $d_k = i$, $d_{k+1} = j$. Następujący warunek wyraża opłacalność zamiany kolejności k-tej i $(k+1)$-szej czynności w uszeregowaniu $d_1, d_2, \ldots, d_n$:

$$\overbrace{a_i s + b_i}^{h_i(s)} + \overbrace{a_j (s + a_i s + b_i) + b_j}^{h_j(s + h_i(s))} \le \overbrace{a_j s + b_j}^{h_i(s)} + \overbrace{a_i (s + a_j s + b_j) + b_i}^{h_i(s + h_j(s))}$$

Po wykonaniu prostych rachunków nierówność ta sprowadza się do następującego prostego kryterium:

$$a_j b_i - a_i b_j \le 0.$$

Powyżej, symbol s oznacza czas rozpoczęcia k-tej czynności. Warto zwrócić uwagę, że warunek opłacalności zamiany dwóch czynności w danym uszeregowaniu nie zależy w ogóle od chwili s. Zdefiniujmy następującą relację między czynnościami:

$$i \,\rho\, j \text{ wtedy i tylko wtedy, gdy } a_j b_i - a_i b_j \le 0 \qquad (1)$$

Możemy teraz sformułować warunek gwarantujący, że dane uszeregowanie czynności $d_1, d_2, \ldots, d_n$ jest optymalne:

Twierdzenie. *Uszeregowanie $d_1, d_2, \ldots, d_n$ jest optymalne (tzn. daje minimalny łączny czas wykonania wszystkich czynności) wtedy i tylko wtedy, gdy spełnione są następujące relacje:*

$$d_i \,\rho\, d_{i+1}, \quad i = 1, 2, \ldots, n-1 \qquad (2)$$

Dowód. Załóżmy przeciwnie, że w uszeregowaniu $d_1, d_2, \ldots, d_n$ istnieje taka para czynności $d_k = i, d_{k+1} = j$, że $j \,\rho\, i$ ale nie zachodzi $i \,\rho\, j$. Wtedy zamieniając kolejność czynności k-tej i $(k+1)$-szej otrzymamy lepsze uszeregowanie, a więc sprzeczność z założeniem, że kolejność $d_1, d_2, \ldots, d_n$ jest optymalna.

Zauważmy, że każde uszeregowanie spełniające warunek (2) ma taki sam łączny czas wykonania wszystkich czynności, bowiem zamieniając miejscami czynności i oraz j, spełniające równocześnie relacje $i \,\rho\, j$ oraz $j \,\rho\, i$, nie zmieniamy łącznego czasu ich wykonania. Dlatego każde uszeregowanie spełniające ten warunek jest optymalne.

Twierdzenie to daje precyzyjne a jednocześnie łatwe do sprawdzenia kryterium określające postać optymalnego uszeregowania. Aby wygenerować takie najlepsze uszeregowanie wystarczy uporządkować czynności według relacji ρ. Problem szeregowania czynności „starzejących się" liniowo sprowadza się zatem

do problemu sortowania. Zwykłą relację ≤ zastępujemy tutaj zdefiniowaną w (1) relacją ρ.

Można w tym celu posłużyć się dowolnym ze znanych algorytmów sortowania. Najlepsze algorytmy działają w czasie $O(n\log n)$, gdzie n oznacza długość porządkowanego ciągu. Należą do nich na przykład: sortowanie stogowe lub sortowanie przez scalanie. W praktyce, najefektywniejszą metodą sortowania jest algorytm *quicksort*, który co prawda dla „złośliwych" danych działa w czasie $O(n^2)$, ale oczekiwany jego czas działania wynosi $O(n\log n)$, gdzie stała kryjąca się za notacją „duże O" jest stosunkowo mała. Poniżej podajemy szkielet programu, który w tablicy `Opt` znajduje optymalne uszeregowanie czynności. Szczegóły pozostawiamy do uzupełnienia Czytelnikowi. W szczególności, w procedurze `Uporzadkuj` można zaadaptować algorytm *quicksort* opisany w książce: L. Banachowski i A. Kreczmar, *Elementy analizy algorytmów*, WNT, Warszawa 1982, str. 111.

```
program Szeregowanie;
 const
  Nmax=10000;
 type
  TabWsp=array[0..Nmax] of Real;
  TabNr =array[0..Nmax] of Integer;
 var
  Ile:Integer; {liczba czynności}
  a,b:^TabWsp; {współczynniki funkcji „starzenia się"}
  Opt:^TabNr;  {konstruowane optymalne uszeregowanie}

 function Mniejsze(i,j:Integer):Boolean;
  {Sprawdzanie relacji (1).}
 begin
  Mniejsze:=a^[Opt^[j]]*b^[Opt^[i]]<a^[Opt^[i]]*b^[Opt^[j]]
 end; {Mniejsze}

 procedure Uporzadkuj(1,n:Integer);
  {Porzadkuje ciag n czynnosci relacja Mniejsze.}

 procedure WczytajDane;
  {Wczytuje dane z pliku SZE.IN i ustala poczatkowe
   uporzadkowanie czynnosci w tablicy Opt.}
```



```
 procedure WypiszWynik;
  {Wypisuje optymalne uszeregowanie dane w tablicy Opt.}

begin {Szeregowanie}
 WczytajDane;
 Uporzadkuj(1,Ile);
 WypiszWynik
end. {Szeregowanie}
```

Omówienie rozwiązań podanych przez uczniów

Około połowa uczestników finału wykryła kryterium, które jest podstawą wyżej opisanego rozwiązania. Poprawne rozwiązania, które bazowały na porządkowaniu czynności zgodnie z relacją ρ, różniły się użyciem rozmaitych algorytmów sortowania. Znaczna część uczniów wykorzystała biblioteczną (w języku C) procedurę quicksort. Niektórzy uczniowie sami zaprogramowali procedurę sortowania. Niestety, wielu z nich użyło algorytmów działających w czasie $O(n^2)$ (sortowanie bąbelkowe, przez wstawianie itp.), co dla dłuższych ciągów czynności drastycznie wydłużało czas działania ich rozwiązań w porównaniu z algorytmami o złożoności $O(n\log n)$. Jedno z rozwiązań wykorzystywało algorytm sortowania przez scalanie, którego pesymistyczny czas działania wynosi również $O(n\log n)$.

Zauważmy, że warunek w definicji relacji $i\,\rho\,j$ można również zapisać następująco:

$$\frac{b_i}{a_i} \le \frac{b_j}{a_j}\ .$$

Ta postać warunku jest bardzo intuicyjna, lecz traci sens, gdy $a_i = 0$ lub $a_j = 0$. Kilku uczestników wpadło w tę pułapkę – ich programy w takich sytuacjach próbowały wykonać dzielenie przez 0.

Choć wielu uczniów odkryło właściwe kryterium uporządkowania czynności, tylko nieliczni potrafili precyzyjnie uzasadnić jego poprawność. Zwykle powoływali się na „intuicję" oraz eksperymentalne sprawdzanie przyjętej hipotezy na danych o niewielkich rozmiarach. W kilku przypadkach „nieświadomie" sortowano czynności według porządku ρ, przyjmując zasadę zamiany sąsiednich czynności w danym uporządkowaniu, o ile ta zamiana jest korzystna. Algorytm taki, polegający na poprawianiu uszeregowania aż do skutku, tj. do momentu gdy nie da się zmniejszyć czasu wykonania poprzez zamianę kolejności dwóch sąsiadujących czynności, daje poprawne rozwiązanie w czasie $O(n^2)$ (wersja sortowania bąbelkowego).

Nie wszyscy finaliści dostrzegli, że do wyszukania optymalnego uszeregowania nie jest potrzebne obliczanie czasu wykonania czynności. Dla dużych zestawów czynności, czas ten zwykle przekraczał zakres wszelkich typów numerycznych dostępnych w językach programowania. Błędne działanie programów obliczających konkretne czasy wykonania czynności dla różnych uszeregowań było zatem spowodowane przekraczaniem zakresu typów numerycznych. Rozwiązania polegające na obliczaniu czasu wykonania czynności dla różnych uszeregowań i wybraniu uszeregowania, dla którego ten czas jest najmniejszy, były więc niezadowalające z co najmniej dwóch powodów:

- ogromnej liczby możliwych uszeregowań (n !),
- błędów przekroczenia zakresu.

Najprawdopodobniej, błędy w rachunkach lub zbyt zgrubne oszacowania czasu wykonywania czynności dla konkretnego uszeregowania, doprowadziły niektórych uczniów do błędnych kryteriów uporządkowania czynności. Na przykład jeden z zawodników mylnie wywnioskował, że składniki b_i nie wpływają w istotny sposób na optymalne uszeregowanie czynności i sortował czynności według współczynników a_i (malejąco).

Pojawiły się także niepoprawne rozwiązania, które konstruowały uszeregowanie zadań metodą zachłanną. Polega ona na tym, że wybieramy kolejne zadania stosując jakąś zasadę lokalnej optymalności. Na przykład, ustalając następną czynność do wykonania, wybieramy tę spośród jeszcze nie wybranych, która zajmie najmniej czasu, jeśli zaczniemy ją wykonywać w chwili zakończenia wykonywania dotychczas wybranych czynności. Aby przekonać się, że strategia ta daje błędne rozwiązania, wystarczy przeanalizować następujący prosty przykład (dla uproszczenia rachunków, liczby w tym przykładzie są całkowite i nie spełniają wymagania, aby $0 \le a_i \le 1$, $0 \le b_i \le 1$):

$a_1 = 4, b_1 = 2, a_2 = 2, b_2 = 3, a_3 = 1, b_3 = 1.$

Metodą zachłanną otrzymujemy uszeregowanie: 3,2,1 o łącznym czasie wykonania wszystkich czynności 32, podczas gdy optymalnym uszeregowaniem jest: 1, 3, 2 o łącznym czasie wykonania zaledwie 18.

Testy

Testy, które służyły do oceny rozwiązań, koncentrowały się na badaniu efektywności (tj. czasu działania oraz zajmowanej pamięci) użytych algorytmów. Cała seria testów o rosnącej wielkości, umożliwiała oszacowanie szybkości przyrostu czasu działania programów ze względu na długość danych wejściowych. Wszystkie testy umożliwiały również wykrycie co najmniej jednego rodzaju błędnych (tj. generujących nieoptymalne uszeregowania) rozwiązań:

- używających strategii zachłannej,

- sortujących według niewłaściwego porządku,
- niepotrzebnie obliczających czasy wykonywania czynności,
- dzielących przez zero, jeśli któryś ze współczynników był równy 0.

8. SPRAWDZANIE I OCENA ROZWIĄZAŃ ZADAŃ OLIMPIADY INFORMATYCZNEJ

W rozdziale tym opisujemy, jak przebiega obecnie proces testowania i oceny rozwiązań zadań w kolejnych etapach Olimpiady. Informacje tutaj zawarte uzupełniają omówienie sprawdzania i oceniania prac, zamieszczone w materiałach z I Olimpiady – zob. *I-OI*, str. 126-132. Podajemy również kilka praktycznych rad, pozwalających unikać prostych błędów, które utrudniają właściwą ocenę rozwiązań lub dyskwalifikują rozwiązania.

Podstawową częścią rozwiązania każdego zadania jest program komputerowy. Ocena poprawności i efektywności działania programu opiera się na wynikach automatycznego testowania. Pomysł takiego sprawdzania rozwiązań powstał na początku Olimpiady. Obecny kształt procesu sprawdzania został znacznie udoskonalony dzięki doświadczeniom zebranym w czasie zawodów I Olimpiady.

Cały proces sprawdzania składa się z trzech (w zawodach I stopnia) lub czterech (w zawodach II i III stopnia) części:

1. Przygotowanie do sprawdzania programów znajdujących się na dyskietkach.
2. Testowanie programów.
3. Analiza wyników testowania i ocena rozwiązań.
4. W przypadku zawodów II i III stopnia – dodatkowo odbywają się indywidualne rozmowy z zawodnikami.

8.1. Przygotowanie programów do sprawdzania

Na początku, rozwiązania znajdujące się na dyskietkach (nadesłanych przez uczniów w zawodach I stopnia lub zebrane od zawodników danego dnia w zawodach II i III stopnia) są przegrywane do komputera i archiwizowane. Rozwiązania każdego uczestnika są jednoznacznie oznaczane numerem i w całym procesie sprawdzania występują pod tym numerem – anonimowo.

Czasem zdarza się, że nadesłane dyskietki są zawirusowane lub uszkodzone. O ile z wirusami na ogół można sobie poradzić, o tyle uszkodzenie dyskietki

najczęściej trwale uniemożliwia odczytanie z niej rozwiązań. Dlatego zaleca się, aby przed wysłaniem dyskietki z rozwiązaniami sprawdzić czy nie zawiera wirusów i spróbować ją odczytać na innym komputerze.

8.2. Testowanie programów

W drugiej części procesu sprawdzania, równolegle na kilku komputerach, członkowie jury testują rozwiązania zadań. Wszystkie komputery użyte do testowania mają te same (lub bardzo zbliżone) parametry techniczne. W sytuacjach, gdy system sprawdzający nie radzi sobie z rozwiązaniem, ingeruje ktoś z jury. Ponadto są wychwytywane również wśród rozwiązań programy interakcyjne, wymagające odpowiedzi lub interwencji z klawiatury.

Programy są testowane w postaci skompilowanej. Jeśli rozwiązanie zawiera tylko program w postaci źródłowej lub skompilowany program ma inną nazwę niż podano w treści zadania, system sprawdzający sygnalizuje osobie nadzorującej konieczność skompilowania i/lub zmiany nazwy programu. Czasem jednak ta dodatkowa kompilacja okazuje się niekorzystna dla zawodnika. Wynika to stąd, że programy są kompilowane przez jury przy standardowym ustawieniu opcji kompilatora. Jeśli w treści programu nie ma odpowiednich dyrektyw, to może się okazać, że te opcje różnią się od opcji użytych przez zawodnika. Należy więc pamiętać o przesyłaniu i oddawaniu programów w postaci skompilowanej, a także o umieszczaniu w treści programu dyrektyw ustalających istotne opcje kompilatora. Z drugiej strony, nie należy zapominać o dołączaniu do rozwiązań zadań w zawodach I stopnia tekstów źródłowych programów. Nie chodzi tutaj tylko o wymóg formalny regulaminu Olimpiady, ale w przypadku np. uszkodzenia na dyskietce skompilowanego programu można ponownie skompilować tekst źródłowy.

Każdy test sprawdzający rozwiązanie ma postać pliku z danymi wejściowymi. Z kolei każdy program, będącego rozwiązaniem zadania, tworzy na podstawie danych wejściowych plik zawierający wyniki. Częstym błędem rozwiązań jest szukanie pliku z danymi wejściowymi poza bieżącym katalogiem (np. w katalogu głównym) lub pod inną nazwą niż nazwa podana w treści zadania. Takie programy nie znajdują danych wejściowych i nie działają poprawnie. Podobnie ważne jest, aby plik z wynikami był tworzony w bieżącym katalogu i miał taką nazwę, jaką podano w treści zadania – w przeciwnym razie system sprawdzający nie może znaleźć wyników. Istotne również jest to, aby przestrzegać ściśle formy zapisu danych wejściowych i wyników. Programy wymagające niestandardowej postaci danych wejściowych dają na ogół złe wyniki lub nie działają. W przypadku niezrozumiałej postaci pliku z wynikami, system sprawdzający wzywa pomocy – wtedy osoba nadzorująca testowanie poprawia postać

zapisu wyników lub uznaje, że wynik jest nieczytelny. Nieczytelność wyników jest najczęściej spowodowana przez:

- przekroczeniem zakresu liczb całkowitych (wtedy zamiast dużych liczb całkowitych pojawiają się na ogół liczby ujemne);
- pominięcie separatorów (wtedy ciąg liczb zlewał się w jedną dużą liczbę);
- pominięcie części danych, jakie powinny znaleźć się w wyniku.

W takich przypadkach nie jest wiadomo, jak poprawić wynik.

Mierzony jest również czas działania programów. Dla każdego testu jest określony maksymalny czas, jaki program zawodnika ma przydzielony na obliczenie wyniku. Jeśli czas ten zostaje przekroczony, to program jest przerywany bez możliwości zaliczenia testu.

Należy pamiętać, aby programy nie podejmowały żadnych prób interakcji z osobą sprawdzającą – jedyną interwencją osób nadzorujących testowanie jest na ogół naciskanie klawisza Enter. W przypadku programów czekających na naciśnięcie jakiegokolwiek klawisza przed rozpoczęciem lub po zakończeniu obliczeń, taka reakcja jest wystarczająca, powoduje jednak pewne opóźnienie. Programy wymagające podania jakiejś informacji (np. nazwy pliku z danymi wejściowymi lub wręcz podania danych wejściowych z klawiatury) są zupełnie bezradne i nie zaliczają żadnych testów. Podobne są konsekwencje wypisywania wyniku na ekranie zamiast do pliku, gdyż system testujący nie uwzględnia zawartości pliku ekranu. Wypisanie wyniku na ekranie, równocześnie z zapisaniem go w pliku lub ilustrowanie przebiegu obliczeń na ekranie są mniejszym błędem. Dodatkowe informacje (wyniki) wysyłane na ekran, chociaż czasem są przyjemne dla oka, nie polepszają jednak wyników testów, a tylko spowalniają pracę programów. W skrajnych przypadkach program może stracić większość czasu na wypisanie różnych informacji na ekranie i w efekcie nie zaliczyć niektórych testów. Najlepiej więc, aby program nie wymagał naciskania żadnych klawiszy i nic nie pisał na ekranie.

Na tym etapie sprawdzania są gromadzone wyniki testów, bez ustalania jeszcze konkretnych ocen (punktowych). Dla każdego testu są zapamiętywane następujące informacje:

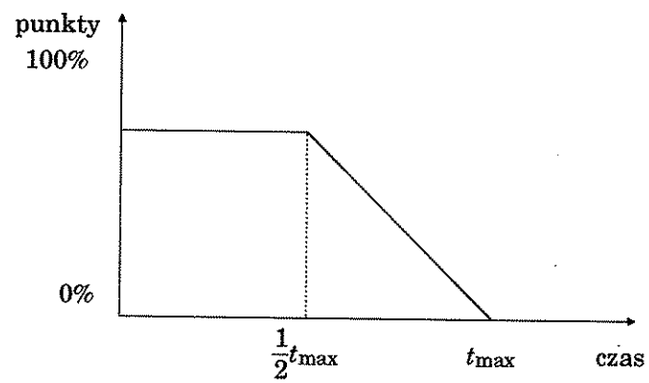
- czas działania programu oraz to, czy program zmieścił się w limicie czasu,
- czy testowany program utworzył plik z wynikami i czy plik ten zawierał jakąkolwiek treść,
- czy wynik był całkowicie czytelny, wymagał korekty, czy też był nieczytelny,
- czy wynik był poprawny, czy też nie.

Każde rozwiązanie jest testowane na kilku lub kilkunastu testach. Stosowane są dwa rodzaje testów: poprawnościowe i wydajnościowe. Testy poprawnościowe są zwykle niewielkie i badają, czy testowany program generuje

poprawny wynik. Zawierają one zarówno typowe dane, jak i dane złośliwe (tj. umyślnie utrudniające rozwiązanie zadania) oraz przypadki graniczne. Testy wydajnościowe mają rozmiary od stosunkowo małych do bardzo dużych. Ich celem jest badanie efektywności rozwiązań, tzn. ile czasu potrzebuje program na obliczenie wyniku dla danych określonej wielkości. Zawierają one zwykle dane losowe lub dane złośliwe, wymagające wykonania dłuższych obliczeń. Wynika stąd, że jedynie poprawne i efektywne rozwiązania mogą z pełnym sukcesem przejść przez wszystkie testy. Przy tym testy wydajnościowe oraz maksymalne czasy obliczeń, jakie programy mają na ich wykonanie są tak dobrane, że użyty język programowania, opcje kompilatora czy niskopoziomowe sztuczki w programach odgrywają drugorzędną rolę w stosunku do zastosowanego algorytmu. Oprócz algorytmu ważne jest również użycie właściwych struktur danych, które zapewniają, że dla danych o maksymalnej wielkości przewidzianej w treści zadania obliczenia nie zostają przerwane z powodu braku pamięci.

8.3. Analiza wyników testowania i ocena rozwiązań

Po zakończeniu testowania wszystkich programów analizowane są wyniki testów. Bada się, ile rozwiązań zaliczyło dany test i jaki był czas obliczeń oraz jak wyniki poszczególnych testów są ze sobą powiązane (tj. czy różne testy nie sprawdzały *de facto* tego samego). Na podstawie wyników takiej analizy statystycznej ustala się ile punktów można zdobyć za dany test. Punkty za test uzyskują tylko te rozwiązania, które zmieściły się w czasie i zapisały poprawne wyniki do pliku. Liczba punktów za test, zależy również od czasu działania programu – tę zależność przedstawono na rysunku 1, na którym $t_{\max}$ oznacza maksymalny czas, przez jaki program mógł działać.



Rysunek 1. Zależność liczby punktów od czasu działania programu.

Jeśli program wykorzystuje połowę przydzielonego na obliczenia czasu lub mniej, to otrzymuje 100% punktów za test. Jeśli działa dłużej, otrzymuje odpowiednio mniej punktów. Sytuacja taka zdarzała się dotychczas jednak rzadko – przeważnie programy albo kończyły działanie w czasie $t_{max}/2$, albo nie mieściły się w czasie t_{max}.

Punkty uzyskane za poszczególne testy są sumowane. W zawodach I i II stopnia suma maksymalnych ocen za testy wynosiła dla każdego zadania 95 punktów, a w zawodach III stopnia – 89 punktów. Ponadto, jeśli wszystkie pliki wynikowe utworzone przez program były całkowicie czytelne (i przynajmniej jeden taki plik był utworzony), to rozwiązanie uzyskiwało dodatkowo 5 punktów. W zawodach II i III stopnia są przyznawane również tzw. punkty uznaniowe – do 6-ciu punktów za każde zadanie.

Tak więc łącznie można uzyskać do 100 punktów za każde zadanie. Punkty uznaniowe są przyznawane na podstawie rozmowy z zawodnikiem, opisu rozwiązania oraz analizy wyników testowania. W II Olimpiadzie Informatycznej Jury zdecydowało, a Komitet Główny zatwierdził tę decyzję, przyznać punkty uznaniowe tylko za zadanie POCHYLNIA – uhonorowane zostały rozwiązania, które wyróżniły się mniejszą liczbą wygenerowanych przestawień beczek.

Mamy nadzieję, że ten opis procesu sprawdzania oraz wskazówki pomogą uczniom w lepszym przygotowywaniu rozwiązań zadań w następnych Olimpiadach Informatycznych.

9. TEKSTY ZADAŃ Z ZAWODÓW MIĘDZYNARODOWYCH

9.1. Zadania Olimpiady Informatycznej Centralnej Europy

9.1.1. Zadanie PRÓŻNUJĄCY KOMPUTER

Komputer notuje czas rozpoczęcia i zakończenia każdego programu aplikacyjnego licząc momenty w skali co 1/100 sekundy. Sprawozdanie dzienne jest zbiorem par (a, b) nieujemnych liczb całkowitych, takich że $0 < a \le b < 8640000$. Para (a, b) oznacza, że program rozpoczął działanie w momencie a i zakończył w momencie b. Momenty a i b zalicza się do czasu działania. Dowolna liczba programów może działać równolegle.

Napisz program, który oblicza

a) długość $b - a + 1$ największego domkniętego przedziału czasowego $[a, b]$, w którym komputer nie wykonywał żadnego programu aplikacyjnego (tzn. był wolny), oraz

b) długość $d - c + 1$ największego domkniętego przedziału czasowego $[c, d]$, w którym komputer wykonywał przynajmniej jeden program aplikacyjny.

Moment t należy do czasu wolnego, jeżeli żaden program aplikacyjny nie działa w momencie t.

Dane wejściowe

Pierwszy wiersz pliku tekstowego DAY1A.DAT zawiera N, liczbę przedziałów ($1 \le n \le 1000$). Każdy z pozostałych N wierszy tego pliku zawiera dwie liczby całkowite nieujemne, momenty rozpoczęcia i zakończenia działania jakiegoś programu aplikacyjnego.

Przykład danych wejściowych

```
9
30000 35000
10000 20000
15000 16000
40000 44000
```

77000 220000
13000 41000
60000 67000
50000 55000
65000 70000

Dane wyjściowe

Plik tekstowy DAY1A.SOL ma zawierać następujące dwie liczby całkowite w następującej kolejności:

a) długość $b - a + 1$ największego przedziału czasowego $[a, b]$, w którym komputer nie wykonywał żadnego programu aplikacyjnego (tzn. był wolny).

b) długość $d - c + 1$ największego przedziału czasowego $[c, d]$, w którym komputer wykonywał przynajmniej jeden program aplikacyjny.

Przykład danych wyjściowych

Dla naszego przykładu danych wejściowych plik DAY1A.SOL będzie zawierał następujące dwa wiersze:

8419999
143001

Punktacja

Maksymalna liczba punktów: 25

Limit czasu: 30 sekund na każdy test

9.1.2. Zadanie ŁAŃCUCH PRZEKAZYWANIA WIADOMOŚCI

Uczniowie pewnej klasy postanowili utworzyć łańcuch przekazywania wiadomości. Każdy uczeń wybiera sobie jednego następcę, któremu będzie bezpośrednio przekazywał otrzymywane wiadomości. Gdy tylko uczeń otrzyma wiadomość, przekazuje ją swemu następcy.

Taki układ nazwiemy łańcuchem przekazywania wiadomości, gdy jest spełniony następujący warunek. Przypuśćmy, że ktoś przekazuje wiadomość swemu następcy, który z kolei przekazuje tę wiadomość swemu następcy itd. Ta wiadomość ostatecznie dotrze do każdego ucznia, łącznie z pierwszym nadawcą tej wiadomości, który zapoczątkował przekazywanie.

Oczywiście nie każdy układ jest łańcuchem przekazywania wiadomości w tym sensie. Napisz program, który dla dowolnego układu opisanego danymi wejściowymi oblicza najmniejszą liczbę koniecznych zmian, które przekształcą ten układ wejściowy w łańcuch przekazywania wiadomości.

Dane wejściowe

Pierwszy wiersz pliku tekstowego DAY1B.DAT zawiera N, liczbę uczniów ($1 \le N < 256$). Każdy z następnych N wierszy zawiera dwa imiona (napisy) przedzielone znakiem >. Po drugim imieniu następuje znak końca wiersza. Imiona mogą mieć długość co najwyżej 20 znaków. Wiersz postaci $A > B$ oznacza, że następcą ucznia A jest B, tzn. A przekazuje wiadomości bezpośrednio uczniowi B.

Przykład danych wejściowych

10
ANITA>PETER
ANDREW>JULIA
DAVID>ANDREW
NATALIE>GABRIELLA
EDITH>DAVID
PETER>ANITA
GABRIELLA>JULIUS
ADAM>DAVID
JULIA>GABRIELLA
JULIUS>JULIA

Dane wyjściowe

Zapisz minimalną liczbę koniecznych modyfikacji w pliku tekstowym DAY1B.SOL jako liczbę całkowitą.

Przykład danych wyjściowych

Dla naszego przykładu danych wejściowych będzie to:

4

Punktacja:

Maksymalna liczba punktów: 35

Limit czasu: 30 sekund na test

9.1.3. Zadanie SPRAWIEDLIWY PODZIAŁ

Dwaj bracia, Alan i Bob, chcą podzielić się prezentami. Każdy prezent powinien przypaść albo Alanowi, albo Bobowi i żadnego prezentu nie da się podzielić. Każdy prezent ma wartość, która jest dodatnią liczbą całkowitą. Niech A i B oznaczają odpowiednio sumy wartości prezentów otrzymanych przez Alana

i Boba. Trzeba zminimalizować bezwzględną wartość różnicy $A - B$. Napisz program, który oblicza wartości A i B.

Dane wejściowe

Pierwszy wiersz pliku tekstowego DAY1C.DAT zawiera N, liczbę prezentów ($1 \le N \le 100$). Pozostała część danych zawiera N liczb całkowitych dodatnich, które są wartościami prezentów. Każda wartość ≤ 200.

Przykład danych wejściowych

```
7
28 7 11 8 9 7 27
```

Dane wyjściowe

Zapisz dwie liczby całkowite A i B w pliku tekstowym DAY1C.SOL

Przykład danych wyjściowych

Dla naszego przykładu danych wejściowych będzie to:

```
48 49
```

Punktacja

Maksymalna liczba punktów: 40

Limit czasu: 30 sekund na test.

9.1.4. Zadanie OGRÓD

W ogrodzie jest N drzew. Ogród ma kształt kwadratu o boku 1000 m. Szukamy prostokąta o największej powierzchni, który nie zawiera w swoim wnętrzu drzew. Boki prostokąta muszą być równoległe do odpowiednich boków ogrodu. Każdy bok prostokąta może zawierać drzewa i może leżeć na boku ogrodu. Położenia drzew są wyznaczone współrzędnymi (x, y) mierzonymi w metrach. Drzewa traktuje się jako punkty, bez żadnych wymiarów.

Początkiem układu współrzędnych jest lewy dolny róg ogrodu i osie są równoległe do boków.

Dane wejściowe

Pierwszy wiersz pliku tekstowego DAY2A.DAT zawiera N, liczbę drzew ($1 \le N \le 600$). W każdym z następnych N wierszy znajdują się całkowite współrzędne (x, y), $0 < x < 1000$, $0 < y < 1000$, określające położenia drzew.

Przykład danych wejściowych

```
7
280 100
200 600
400 200
135 800
800 400
600 800
900 210
```

Dane wyjściowe

Zapisz powierzchnię szukanego prostokąta jako liczbę całkowitą w pliku tekstowym DAY2A.SOL.

Przykład danych wyjściowych

Dla naszego przykładu danych wejściowych będzie to:

```
360000
```

Punktacja

Maksymalna liczba punktów: 30

Limit czasu: 30 sekund na każdy test

9.1.5. Zadanie PRZYJĘCIE

Prezes firmy organizuje przyjęcie dla swych pracowników. Firma ma strukturę hierarchiczną; relacja podległości tworzy drzewo, którego korzeniem jest prezes. Aby wszyscy obecni mogli się dobrze bawić, prezes zarządził, by żaden pracownik nie siedział przy tym samym stole ze swoim zwierzchnikiem.

Zadanie polega na obliczeniu najmniejszej liczby stołów, niezbędnej do rozsadzenia gości zgodnie z zarządzeniem prezesa. Dane wejściowe programu opisują relację bezpośredniej podległości w firmie. Relacja podległości jest określona przez relację bezpośredniej podległości w następujący sposób: osoba P jest zwierzchnikiem osoby Q, jeśli P jest bezpośrednim zwierzchnikiem Q albo istnieją osoby $P_1, \ldots, P_k$ $(1 < k)$ takie, że $P = P_1$, $Q = P_k$ oraz P_i jest bezpośrednim zwierzchnikiem P_{i+1} $(i = 1, \ldots, k - 1)$.

Liczba uczestników wynosi N $(1 \le N \le 200)$ i przy każdym stole jest T miejsc $(2 \le T \le 10)$.

Dane wejściowe

Pracownicy firmy są pooznaczani za pomocą początkowych N ($1 \le N \le 200$) liczb naturalnych. Prezes jest oznaczony numerem 1 i nie ma zwierzchnika. Pierwszy wiersz pliku tekstowego DAY2B.DAT zawiera T, liczbę miejsc przy stole ($2 \le T \le 10$). Drugi wiersz zawiera liczbę N ($1 \le N \le 200$) uczestników; każdy pracownik o numerze od 1 do N weźmie udział w przyjęciu. W trzecim wierszu znajduje się N liczb: i-ta liczba w tym wierszu oznacza bezpośredniego zwierzchnika i-tej osoby. Pierwszą liczbą w tym wierszu jest 0, co wskazuje, że prezes nie ma zwierzchnika.

Przykład danych wejściowych

```
4
13
0 1 9 9 9 2 2 1 1 7 8 8 10
```

Dane wyjściowe

Zapisz w pliku tekstowym DAY2B.SOL najmniejszą liczbę stołów niezbędych do rozmieszczenia wszystkich uczestników przyjęcia w podany wyżej sposób.

Przykład danych wyjściowych

Dla naszego przykładu danych wejściowych będzie to:

```
5
```

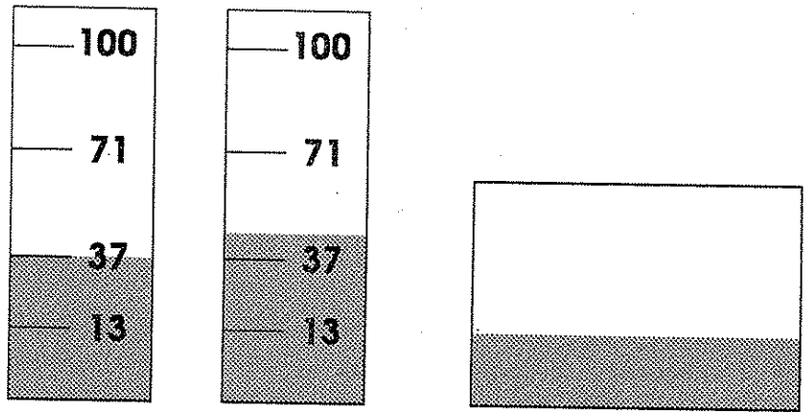
Punktacja

Maksymalna liczba punktów: 30
Limit czasu: 30 sekund na każdy test

9.1.6. Zadanie MENZURKI

Mamy trzy szklane naczynia. Pojemność każdego wynosi 100 jednostek (cm^3). Pierwsze dwa naczynia mają kreski, jednakowe na obu naczyniach, każda wskazuje objętość, mierzoną od dna do kreski. Ta objętość jest zapisana obok kreski (zob. rysunek). Na początku pierwsze naczynie zawiera 100 jednostek objętości płynu, pozostałe są puste.

Napisz program, który rozstrzyga, czy da się wyodrębnić w trzecim naczyniu jedną jednostkę objętości i jeśli tak, to oblicza najmniejszą liczbę kroków, w której to można zrobić. W każdym kroku po wykonaniu operacji przynajmniej jedno z użytych naczyń powinno zawierać płyn do kreski albo być puste. W czasie tego postępowania możemy śledzić zmiany zawartości wszystkich trzech naczyń.



Dane wejściowe

Pierwszy wiersz pliku tekstowego DAY2C.DAT zawiera N, czyli liczbę kresek na pierwszych dwu naczyniach ($1 \le N \le 20$). Pozostała część pliku zawiera N liczb całkowitych w porządku wzrastającym, które są objętościami oznaczonymi kreskami na dwóch naczyniach. Dla każdej kreski objętość V spełnia nierówność $1 \le V \le 100$ i największą wartością jest 100.

Przykład danych wejściowych

```
4
13 37 71 100
```

Dane wyjściowe

Zapisz ciąg znaków YES w pierwszym wierszu pliku tekstowego DAY2C.SOL, jeśli da się wyodrębnić jedną jednostkę objętości w trzecim naczyniu lub zapisz NO w przeciwnym przypadku. Jeśli odpowiedź jest pozytywna, to napisz w drugim wierszu najmniejszą liczbę kroków.

Przykład danych wyjściowych

Dla naszego przykładu danych wejściowych będzie to:

```
YES
8
```

Punktacja:

Maksymalna liczba punktów: 40
Limit czasu: 30 sekund na każdy test

9.2. Zadania VII Międzynarodowej Olimpiady Informatycznej

9.2.1. Zadania wstępne

Zadanie 1

W skomplikowanym projekcie technicznym uczestniczy 100 naukowców, którzy szczycą się swoimi uzdolnieniami w dziedzinie rachunku pamięciowego. W małym pokoju na tablicy napisano liczbę zero. Zadanie każdego ze stu naukowców polega na wykonaniu 100 razy następującej procedury.

1. Wejdź do pokoju.
2. Przeczytaj i zapamiętaj liczbę zapisaną na tablicy.
3. Wyjdź z pokoju.
4. Dodaj w pamięci 37 do zapamiętanej liczby.
5. Wejdź do pokoju.
6. Wytrzyj liczbę zapisaną na tablicy.
7. Zapisz na tablicy wynik wykonanego w pamięci dodawania.
8. Wyjdź z pokoju.

Każdy naukowiec pracuje we własnym tempie, niezależnie od pozostałych. Pokój jest tak mały, że w dowolnym momencie może się w nim znajdować co najwyżej jeden naukowiec.

Największą liczbą, jaka może zostać zapisana na tablicy, po zakończeniu pracy przez wszystkich naukowców jest 370000. Jaka może być najmniejsza końcowa wartość zapisana na tablicy?

Analiza i rozwiązanie Zadania 1

Zadanie ilustruje jedną z trudności związanych z tzw. obliczeniami równoległymi, kiedy wiele procesorów przetwarza równocześnie te same dane. Każdy z naukowców reprezentuje procesor; a dane to liczba na tablicy. Tablicę możemy traktować jako **wspólną zmienną** x, ponieważ ma do niej dostęp wiele procesorów, kóre mogą odczytywać i zmieniać jej wartość.

Program wykonywany przez każdy z procesorów składa się ze stu powtórzeń przypisania zmiennej jej wartości, powiększonej o 37. Programy wszystkich procesorów razem stanowią tzw. program równoległy. Wykonanie tego równoległego programu polega na zwiększeniu 10000 razy wartości zmiennej x.

W programach równoległych dostęp do wspólnej zmiennej musi być kontrolowany. Na przykład, co się stanie, jeśli dwa procesory równocześnie będą zmieniały wartość wspólnej zmiennej? Wynik może być bezwartościowy. W naszym przypadku byłoby to niemożliwe, ponieważ małe rozmiary pokoju zapewniają wzajemne wykluczanie: tylko jeden naukowiec może w danym momencie wejść do pokoju i zmienić zapis na tablicy.

W przypadku programu sekwencyjnego (wykonywanego przez jeden procesor), zwiększenie początkowej wartości zero 10000 razy o 37 daje wynik 370000. Równoległe wykonywanie takiej samej operacji może jednak doprowadzić do innego (mniejszego) wyniku, ponieważ zwiększanie wartości zmiennej można podzielić na trzy operacje (odczytanie, zwiększenie zapamiętanej wartości, zapisanie). Na przykład naukowiec $N[0]$ odczytuje zero, naukowiec $N[1]$ również odczytuje zero, obaj zwiększają zapamiętane zera o 37, następnie $N[0]$ zapisuje swój wynik 37 i $N[1]$ zapisuje swoje 37. W ten sposób dwukrotne zwiększenie wartości daje taki sam efekt jak jedno. Oczywiście również 100 zwiększeń może mieć taki sam skutek jak jedno zwiększenie, a tym samym innym możliwym końcowym wynikiem zwiększenia wartości na tablicy 10000 razy o 37 może być liczba 3700.

To jednak jeszcze nie koniec historii. Wnikliwa analiza różnych możliwości wskazuje, że ostatecznym wynikiem może być każda wielokrotność liczby 37 od 74 do 370000.

A oto jeden z możliwych przebiegów procesu, kończący się wynikiem 74.

1. Wszyscy naukowcy odczytują i zapamiętują zero.
2. Naukowiec $N[0]$ wykonuje swoją procedurę zwiększania liczby na tablicy 99 razy i pozostawia na tablicy liczbę 3673.
3. Naukowiec $N[1]$ wykonuje całe swoje zadanie do końca, a następnie po kolei $N[2]$ aż do $N[98]$ wykonują do końca swoje zadania. Na tablicy pozostaje liczba 3700, bo każdy kolejno zaczyna od zera.
4. Naukowiec $N[99]$ kończy pierwsze zwiększenie (ma w pamięci zero) i zostawia na tablicy 37.
5. Naukowiec $N[0]$ odczytuje z tablicy i zapamiętuje liczbę 37.
6. $N[99]$ wykonuje zadanie do końca zwiększając liczbę na tablicy 99 razy i zostawia 3700.
7. $N[0]$ zwiększa zapamietaną liczbę 37 o 37 i zapisuje na tablicy ostateczny wynik 74.

Zadanie 2

Jednym z często wykonywanych zadań jest wyszukiwanie elementu w uporządkowanym ciągu danych. Pomyśl o wyszukiwaniu słowa w słowniku lub

nazwiska w książce telefonicznej. Przedstawiamy poniżej dwa algorytmy, które są różnymi rozwiązaniami problemu wyszukiwania. Masz zaplanować i przeprowadzić odpowiednie eksperymenty, mające na celu porównanie efektywności tych algorytmów.

Załóżmy, że S jest ciągiem N elementów, gdzie $N > 0$. Wyrazy ciągu S: $S[i]$ dla $0 \le i < N$ są uporządkowane rosnąco, tzn. $S[i-1] < S[i]$ dla $0 < i < N$. Dla obu algorytmów dany jest element x występujący w ciągu S, a wynikiem ma być indeks i z zakresu: $0 \le i < N$ taki, że: $S[i] = x$. W obu algorytmach występują zmienne j oraz h.

Algorytm A

1. Ustal wartości początkowe: $i := 0; j := N$.
2. Dopóki $i \neq j - 1$ wykonuj kroki 3a i 3b.
3. (a) Jeśli $i + j$ jest liczbą parzystą, to $h := (i + j)/2$, inaczej $h := (i + j - 1)/2$.
3. (b) Jeśli $x < S[h]$, to $j := h$, inaczej $i := h$.

Nierówności $S[i] \le x \le S[j-1]$ są **niezmiennikiem pętli „dopóki"**, tzn. x występuje stale w obszarze od $S[i]$ do $S[j-1]$. Jeśli ten obszar składa się z jednego elementu ($i = j - 1$), to poszukiwanie się kończy (patrz krok 2). W przeciwnym przypadku dzielimy obszar na dwie części, nadając h wartość w środku pomiędzy i oraz j (krok 3a). Zależnie od tego w której części znajduje się x, zmieniamy i albo j (krok 3b).

Algorytm B

Algorytm B otrzymujemy z algorytmu A przez następujące rozwinięcie kroku 3b.

3. (b) Jeśli $S[h] = x$, to $i := h$ oraz $j := h + 1$
inaczej
jeśli $x < S[h]$, to $j := h$, inaczej $i := h$.

W przypadku algorytmu B poszukiwanie może się zakończyć natychmiast, kiedy okaże się, że element w środku badanego obszaru spełnia $S[h] = x$.

Należy wyjaśnić dwa zagadnienia.

1. Jaka jest różnica średniej liczby powtórzeń kroku 3 w przypadkach A oraz B?
2. Jaka jest różnica czasów wyszukiwania elementu x, jeśli miarą czasu wyszukiwania jest łączna liczba wszyskich operacji elementarnych?

Powinieneś zbadać ciągi o różnych długościach. Możesz zakładać, że poszukiwane wartości x są rozrzucone jednostajnie na całej długości ciągu S, tzn. prawdopodobieństwo, że element $S[i]$ jest poszukiwanym x jest takie samo dla każdego $i < N$. Przypisania i porównania są operacjami elementarnymi. Także wykonanie kroku 3a jest jedną operacją elementarną, ponieważ w większości

komputerów istnieje dla tej operacji specjalny rozkaz (shift). Tak więc w przypadku algorytmu A, wykonanie kroku 1 wymaga dwóch operacji elementarnych: wykonanie kroku 2 to jedna operacja elementarna, wykonanie kroku 3a to też jedna operacja elementarna. Wykonanie kroku 3b polega na wykonaniu dwóch operacji elementarnych (porównania i przypisania).

Analiza i rozwiązanie Zadania 2

Algorytm A to klasyczne wyszukiwanie metodą **dziel i zwyciężaj**, określane w literaturze terminem BINARY SEARCH (wyszukiwanie przez połowienie). Algorytm B jest wariantem algorytmu A. W przypadku algorytmu A liczba powtórzeń jest niezależna od wartości x i wynosi w przybliżeniu $\log_2 N$. W przypadku algorytmu B liczba powtórzeń może być (dla pewnych konkretnych danych) znacznie mniejsza. Jeśli jednak weźmiemy pod uwagę średnią liczbę powtórzeń, to zysk jest minimalny. Kiedy zwiększamy długość badanych ciągów, to różnica przeciętnej liczby powtórzeń dla dwóch algorytmów A oraz B jest zbieżna do 1 (patrz tabela poniżej). Oszczędzamy jedno porównanie, nic więcej. Karą za to „udoskonalenie" jest zwiększona liczba porównań w każdym powtórzeniu. Ostatecznie więc algorytm B okazuje się średnio znacznie wolniejszy.

Mogą to potwierdzić odpowiednie eksperymenty. (Teoretyczna analiza jest możliwa ale żmudna. Łatwo można poradzić sobie z ciągami, kórych długości są potęgami dwójki, ale w innych przypadkach pojawiają się komplikacje). Istnieje wiele możliwości zaplanowania eksperymentu. Naiwne podejście polega na wielokrotnym powtórzeniu następującego eksperymentu. Utwórz losowo uporządkowany alfabetycznie ciąg napisów (o losowych długościach). Zastosuj algorytmy wyszukiwania do losowej liczby losowych wartości x i zmierz rzeczywisty czas wyszukiwania. Po zbadania dostatecznej liczby przypadków zrób odpowiednie zestawienie statystyczne.

Więcej informacji daje następujący eksperyment. Zauważ, że wystarczy badać ciągi kolejnych liczb całkowitych, gdzie $S[i] = i$, ponieważ nie jest istotne *czego* szukamy, a jedynie *jak*.

Badając różne długości ciągów, wywołujemy algorytmy A oraz B *tylko raz* dla każdej dopuszczalnej wartości x, zliczając liczbę powtórzeń oraz wszystkich operacji elementarnych (porównań i przypisań) Po wykonaniu zadania wyszukiwania dla wszystkich możliwych x, dzielimy odpowiednie sumy, przez długość ciągu (równą liczbie eksperymentów) i otrzymujemy odpowiednie średnie. Szeroki zakres możliwych długości ciągów można zbadać wykonując stosunkowo niedużą liczbę eksperymentów, np. badając przypadki kiedy długości są potęgami liczby 10 od 1 do miliona. (Ze względu na naturę algorytmów wyszukiwania, długości będące potęgami dwójki stanowią bardzo szczególny przypadek

i wyniki ich badań nie mogą być uważane za reprezentatywne). Wyniki eksperymentu przedstawia nastepująca tabela:

Długość ciągu	Algorytm A		Algorytm B	
	średnia liczba powtórzeń	średnia liczba operacji	średnia liczba powtórzeń	średnia liczba operacji
1	0.00	3.00	0.00	3.00
10	3.40	16.60	2.80	17.00
100	6.72	29.88	5.79	31.95
1000	9.98	42.90	8.99	47.93
10000	13.36	56.45	12.36	64.81
100000	16.69	69.76	15.69	81.45
1000000	19.95	82.81	18.95	97.76

Zadanie 3

Twój program powienien zacząć od odczytania jednego wiersza ze *standardowego wejścia*. Ten wiersz zawiera liczbę n ciężarówek do posortowania ($1 \le n \le 10$). Ciężarówki są oznaczone od 1 do n i wszystkie różnią się masami. Masy są znane tylko środowisku.

Dalej Twój program powinien kontynuować dialog ze środowiskiem, aby posortować te oznaczenia ciężarówek według mas w porządku wzrastającym. W trakcie dialogu twój program powinien pytać środowisko, która z ciężarówek ma większą masę.

Aby porównać ciężarówki oznaczone i oraz j, Twój program powinien napisać jeden wiersz na *standardowym wyjściu*. Ten wiersz powinien zaczynać się od wielkiej litery C, po której następują i oraz j. Następnie Twój program powinien przeczytać odpowiedź środowiska w postaci jednego wiersza ze *standardowego wejścia*, zawierający oznaczenie cięższej ciężarówki (jeśli $i = j$, to odpowiedzią jest i).

Wreszcie, gdy Twój program ustalił kolejność, powinien wypisać jeden wiersz na *standardowe wyjście*. Ten wiersz powinien zaczynać się od wielkiej litery L, po której następuje n oznaczeń w kolejności wzrastających mas tak oznaczonych ciężarówek.

(Zadanie dodatkowe: uczyń liczbę porównań jak najmniejszą).

Przykład dialogu

Przykładowy dialog dotyczący trzech ciężarówek i czterech porównań (pierwsze nie jest niezbędne) może mieć postać:

Standardowe wejście	*Standardowe wyjście*
	3
C 1 1	1
C 1 2	2
C 2 3	2
C 1 3	1
L 3 1 2	

Wersja wsadowa tego zadania

Pierwszy wiersz pliku INPUT.TXT zawiera liczbę n ciężarówek do posortowania ($1 \le n \le 10$). Każdy z następnych wierszy opisuje jedną ciężarówkę za pomocą napisu złożonego z co najmniej 1 i co najwyżej 20 małych liter (od 'a' do 'z').

Masa ciężarówki jest podana jako długość napisu. Wszystkie ciężarówki różnią się masami.

W pierwszym wierszu pliku OUTPUT.TXT Twój program powinien napisać masę najlżejszej i najcięższej ciężarówki z pliku wejściowego. W następnych n wierszach powinien wypisać napisy wejściowe w kolejności wzrastających mas, jeden napis w wierszu.

Przykład danych wejściowych i wyjściowych

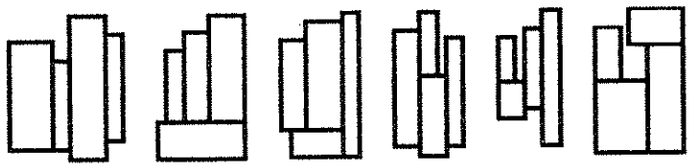
Przykładowy plik wejściowy z czterema ciężarówkami i odpowiadający mu plik wyjściowy:

INPUT.TXT
```
4
aabbcc
klm
z
ns
```

OUTPUT.TXT
```
1 6
z
ns
klm
aabbcc
```

9.2.2. Zadanie PAKOWANIE PROSTOKĄTÓW

Dane są cztery prostokąty. Znajdź najmniejszy **obejmujący** prostokąt, w którym można zmieścić te prostokąty tak, aby nie zachodziły na siebie. Przez najmniejszy prostokąt rozumiemy prostokąt o najmniejszym polu.



Rysunek 1. Sześć podstawowych ułożeń czterech prostokątów.

Boki wszystkich czterech wewnętrznych prostokątów powinny być równoległe do odpowiednich boków obejmującego je prostokąta. Rysunek 1 pokazuje sześć sposobów dopasowania czterech prostokątów. Są to jedyne możliwe podstawowe dopasowania, gdyż każde inne można otrzymać z jednego z nich za pomocą obrotu lub odbicia.

Może istnieć wiele różnych obejmujących prostokątów, które spełniają podane wymagania i mają takie same pola. Musisz znaleźć *wszystkie* takie obejmujące prostokąty.

Dane wejściowe

Plik wejściowy INPUT.TXT ma 4 wiersze. Każdy wiersz opisuje jeden dany prostokąt za pomocą dwóch liczb całkowitych dodatnich: długości boków prostokąta. Każdy bok prostokąta ma długość nie mniejszą niż 1 i nie większą niż 50.

Dane wyjściowe

Plik wyjściowy OUTPUT.TXT powinien zawierać o jeden wiersz więcej niż jest rozwiązań. Pierwszy wiersz zawiera jedną liczbę całkowitą: najmniejsze możliwe pole obejmującego prostokąta (Podzadanie A). Każdy z następnych wierszy zawiera jedno rozwiązanie przedstawione w postaci dwóch liczb p i q, gdzie $p \le q$ (Podzadanie B). Te wiersze powinny być posortowane w rosnącym porządku p i wszystkie muszą być parami różne.

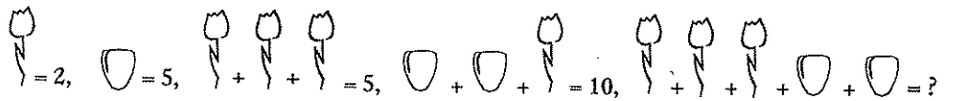
Przykład danych wejściowych i wyjściowych

INPUT.TXT
1 2
2 3
3 4
4 5

OUTPUT.TXT
40
4 10
5 8

9.2.3. Zadanie OFERTY SPECJALNE

W pewnym sklepie towar każdego rodzaju ma swoją cenę. Na przykład, cena kwiatka wynosi 2 ICU (*Informatics Currency Unit*), zaś cena wazonu to 5 ICU. Aby przyciągnąć jak najwięcej klientów, sklep wprowadził oferty specjalne.



Oferta specjalna to jedna lub więcej jednostek towarów sprzedawanych za niższą cenę. Przykłady: trzy kwiatki za 5 ICU zamiast 6, lub dwa wazony i jeden kwiatek za 10 ICU, zamiast 12.

Napisz program, obliczający cenę, którą klient musi zapłacić za pewne zakupy. Należy wykorzystać oferty specjalne jak najlepiej, to znaczy cena powinna być tak niska, jak tylko to jest możliwe. Nie wolno dodawać nowych towarów, nawet jeśli to obniżyłoby cenę. Dla cen i ofert przedstawionych na rysunku najniższa cena za trzy kwiatki i dwa wazony wynosi 14 ICU: dwa wazony i jeden kwiatek za obniżoną cenę 10 ICU oraz dwa kwiatki za zwykłą cenę 4 ICU.

Dane wejściowe

Dane wejściowe są brane z dwu plików INPUT.TXT oraz OFFER.TXT. Pierwszy z tych plików opisuje zakupy (zawarte w „sklepowym koszyku"). Drugi plik opisuje oferty specjalne. W obu plikach występują tylko liczby całkowite.

Pierwszy wiersz pliku INPUT.TXT zawiera liczbę b różnych rodzajów towarów w koszyku ($0 \le b \le 5$). Każdy z następnych b wierszy zawiera trzy wielkości c, k i p. Wartość c jest jednoznacznym kodem towaru ($1 \le c \le 999$). Wartość k oznacza, ile jednostek tego towaru jest w koszyku ($1 \le k \le 5$). Wartość p jest normalną ceną za jednostkę towaru ($1 \le p \le 999$). Zauważ, że w koszyku może być co najwyżej $5 * 5 = 25$ jednostek towarów.

Pierwszy wiersz w pliku OFFER.TXT zawiera liczbę s ofert specjalnych ($0 \le s \le 99$). Każdy z następnych s wierszy opisuje jedną ofertę, podając jej strukturę i jej obniżoną cenę. Pierwsza liczba n w takim wierszu jest liczbą różnych rodzajów towarów wchodzących w skład oferty ($1 \le n \le 5$). Kolejne n par liczb (c, k) wskazuje, że k jednostek towaru ($1 \le k \le 5$) o kodzie c ($1 \le c \le 999$) stanowi składnik danej oferty. Ostatnia liczba p w wierszu jest obniżoną ceną ($1 \le p \le 9999$). Obniżona cena oferty jest niższa niż suma cen normalnych.

Dane wyjściowe

Wpisz do pliku wyjściowego OUTPUT.TXT jeden wiersz zawierający najniższą możliwą cenę za zakupy podane w pliku wejściowym.

Przykład danych wejściowych i wyjściowych

Dla przykładu na rysunku, zawartości plików wejściowych i wyjściowego mają następującą postać. Kodem kwiatka jest 7, a kodem wazonu jest 8.

```
INPUT.TXT
2
7 3 2
8 2 5
OFFER.TXT
2
1 7 3 5
2 7 1 8 2 10
OUTPUT.TXT
14
```

9.2.4. Zadanie DRUKOWANIE

Klient i serwer

Mamy dwóch użytkowników, z których każdy ma komputer. Komputery są identyfikowane za pomocą ich nazw: CLIENT(1) i CLIENT(2). Oba komputery korzystają wspólnie z jednej lub wielu drukarek nazwanych: SERVER(1), SERVER(2) itd. Zadania drukowania z obu komputerów mogą być obsługiwane

jedynie sekwencyjnie. Aby skoordynować komunikację obu komputerów z drukarką użyjemy specjalnych obiektów: semaforów.

Semafor

Z każdą drukarką jest związany jeden semafor. Semafor przyjmuje jeden z dwóch stanów: S1 lub S2. Gdy drukarka jest gotowa na przyjęcie zadania drukowania, stanem semafora jest S1. Jak długo drukarka jest zajęta drukowaniem, stanem semafora jest S2.

Semafor może wykonać dwa rodzaje zmian stanów: 'S1 → S2' i 'S2 → S1'. Gdy użytkownik przesyła do komputera zadanie drukowania, to komputer przesyła komunikat „Are_you_open?" do semafora. Jeśli stanem semafora jest S1, to ten stan semafora zmienia się na S2 i semafor wysyła komunikat „Open" do komputera, który wysłał komunikat „Are_you_open?". Jeśli stanem semafora jest S2, to ten semafor wysyła z powrotem komunikat „Closed". Po zakończeniu drukowania drukarka wysyła temu semaforowi komunikat „Ready". Po otrzymaniu komunikatu „Ready", semafor zmienia swój stan na S1.

Typ obiektu SEMAPHORE

W Dokumentacji 1 znajdziesz specyfikację typu obiektów SEMAPHORE. Ta specyfikacja zawiera dopuszczalne identyfikatory i stany obiektów typu SEMAPHORE, listę priorytetów ('Priority List'), diagram komunikacji ('Communication Diagram'), diagram zmian stanów ('State Transition Diagram') oraz procedury przyjmowania ('Receive Procedures') komunikatów: „Ready" i „Are_you_open?". Procedury przyjmowania opisują, jak semafor reaguje na komunikaty.

Lista priorytetów jest konieczna, ponieważ komunikaty nadchodzące do semafora jednocześnie mogą być przyjmowane tylko sekwencyjnie. Lista priorytetów w Dokumentacji 1 wskazuje, że każdy komunikat z SERVER ma wyższy priorytet niż komunikat z CLIENT i że np. komunikat z SERVER(2) ma wyższy priorytet niż komunikat z SERVER(3).

Typ obiektu CLIENT

W Dokumentacji 2 znajdziesz specyfikację typu obiektów CLIENT. Obiekt typu CLIENT może przyjmować trzy stany: SA, SB lub SC. Stanem klienta jest SA, gdy ten klient nie wysłał zadania drukowania do serwera, ani żaden z serwerów nie jest zajęty obsługą zadania drukowania tego klienta. Klient, który jest w stanie SB, pragnie dostępu do serwera. Klient może otrzymać dostęp do serwera jedynie za pośrednictwem semafora. Stanem klienta jest SC, gdy pewien serwer obsługuje polecenia drukowania tego klienta.

Klient może wykonać trzy rodzaje zmian stanów: 'SA → SB', 'SB → SC', 'SC → SA'. Gdy klient jest w stanie SB, to może odebrać komunikat „Closed" od semafora. Po otrzymaniu tego komunikatu ten klient czeka przez pewien okres, 'Waiting_Period', zanim ponownie wyśle do semafora komunikat „Are_you_open?".

Gdy semafor wyśle komunikat „Open" do tego klienta, ten zmienia swój stan na SC i wysyła zadanie drukowania z komunikatem „S_Job" do serwera związanego z wysyłającym semaforem. Następnie ten serwer obsługuje zadanie drukowania. Po zakończeniu obsługi tego zadania, serwer wysyła równocześnie dwa komunikaty: „Ready" – do swego semafora i „C_Ready" – do obsługiwanego klienta. Ten klient zmienia wówczas swój stan z SC na SA. Możesz założyć, że drukarki są idealne. Zakończą obsługę każdego otrzymanego zadania drukowania. Na przykład, nigdy nie pojawi się sytuacja „brak_papieru".

Komunikacja

W Dokumentacji 3 znajdziesz w diagramie struktury komunikacji ('Communication Structure Diagram') wszystkie typy komunikatów, które można przesyłać pomiędzy różnymi typami obiektów. W liście komunikatów ('Message List') znajdziesz specyfikację każdego typu komunikatu. Każdy komunikat ma identyfikator, nadawcę, odbiorcę i czasami ma treść.

Gdy nadawca posyła komunikat z identyfikatorem A w chwili t, to odbiorca w chwili $t + 1$ obsługuje ten komunikat, wykonując procedurę przyjmowania ('Receive Procedure') A.

Jeśli wielu nadawców posyła komunikaty temu samemu odbiorcy w chwili t, to odbiorca obsłuży wszystkie te komunikaty w chwili $t + 1$, w tej samej kolejności w jakiej nadawcy tych komunikatów są uszeregowani w liście priorytetów ('Priority List') odbiorcy.

Podzadanie A

Sieć Lokalna (*Local Area Network*, LAN) zawiera w chwili 0 następujące obiekty:

Obiekt: CLIENT(1), Client.State = SA,Waiting_Period = 2, Number_of_Servers= 1.
Obiekt: CLIENT(2), Client.State = SA,Waiting_Period = 1, Number_of_Servers= 1.
Obiekt: SERVER(1).
Obiekt: SEMAPHORE(1), Semaphore.State = S1.

W tej sieci LAN wysłano, między innymi, następujące komunikaty:
W chwili 1: CLIENT(1) wysyła komunikat z identyfikatorem „Are_you_open?".
W chwili 2: CLIENT(2) wysyła komunikat z identyfikatorem „Are_you_open?".
W chwili 4: SERVER(1) wysyła komunikat z identyfikatorem „Ready".

W chwili 5: CLIENT(1) wysyła komunikat z identyfikatorem „Are_you_open?".

Dokumentacja 4 wskazuje dla SEMAPHORE(1), CLIENT(1) i CLIENT(2), w rozkładzie sięgającym aż do chwili 6, które komunikaty otrzymują te obiekty i które komunikaty wysyłają, oraz w jakich stanach są i do jakich stanów przechodzą.

Pytanie A.1

Co by się stało gdyby, dodatkowo, CLIENT(1) otrzymał komunikat „C_Job" w chwili 4?

Napisz swą odpowiedź w Dokumentacji 5.

Pytanie A.2

Co by się stało gdyby CLIENT(2), a nie CLIENT(1) otrzymał ten dodatkowy komunikat „C_Job" w chwili 4?

Napisz swą odpowiedź w Dokumentacji 5.

Pytanie A.3

Uzupełnij rozkład z Dokumentacji 4 do chwili 13, zakładając następujące zdarzenia:
W chwili 8: SERVER(1) wysyła komunikat z identyfikatorem „Ready".
W chwili 10: CLIENT(1) otrzymuje komunikat z identyfikatorem „C_Job".
W chwili 12: SERVER(1) wysyła komunikat z identyfikatorem „Ready".

Podzadanie B

Sieć LAN została rozszerzona. Zawiera teraz dwie pary semafor-serwer: (SEMAPHORE(1), SEMAPHORE(2), SERVER(1), SERVER(2)). Dla każdego klienta CLIENT wartość Number_of_Servers wynosi 2.

Aby korzystać z obu drukarek, należy dokonać zmiany w definicji typu obiektów CLIENT opisanej w Dokumentacji 2.

W Dokumentacji 6 znajdziesz zmienione procedury przyjmowania ('Receive Procedures'): C_Job i Wait.

W Dokumentacji 6 znajdziesz również opis sytuacji w chwili 0.

W następujących chwilach są odbierane następujące komunikaty:
W chwili 0: CLIENT(1) otrzymuje komunikat „C_Job" od użytkownika.
W chwili 0: CLIENT(2) otrzymuje komunikat „C_Job" od użytkownika.
W chwili 4: SEMAPHORE(1) otrzymuje komunikat „Ready".

Co się stanie w tej rozszerzonej sieci LAN zgodnie ze zmienioną definicją typu obiektów CLIENT?

Zaznacz poprawne odpowiedzi w Dokumentacji 6.

Podzadanie C

Zmiana definicji typu obiektów CLIENT w podzadaniu B jest nieefektywna dla rozszerzonej sieci LAN z więcej niż jedną parą semafor-serwer.

Podzadanie C.1

Zmień definicję typu obiektów CLIENT (porównaj z Dokumentacją 2) tak, aby sieć LAN z więcej niż jedną parą semafor-serwer mogła funkcjonować następująco:

Obiekt CLIENT(i) może korzystać z dowolnego serwera w sieci LAN, ale obiekt CLIENT(i) może mieć w danej chwili tylko jedno zadanie drukowania obsługiwane przez serwery. Każde zadanie drukowania klienta jest obsługiwane tylko raz.

Obiekt CLIENT(i) wysyła komunikat „Are_you_open?" kolejno do wielu semaforów, aż osiągnięta zostanie pewna graniczna liczba otrzymanych komunikatów „Closed" lub CLIENT(i) otrzyma jeden komunikat „Open".

Gdy obiekt CLIENT(i) osiągnie wartość graniczną, CLIENT(i) odczekuje pewien okres, Waiting_Period, zanim ponownie wyśle serię komunikatów „Are_you_open?". Napisz swoje rozwiązanie w Dokumentacji 7.

Podzadanie C.2

Zmień typ obiektów CLIENT tak, aby obiekt CLIENT(i) mógł mieć również kilka zadań drukowania obsługiwanych jednocześnie przez kilka serwerów. Jednakże liczba zadań drukowania każdego CLIENT(i) powinna być ograniczona przez CLIENT(i).Job_Maximum.

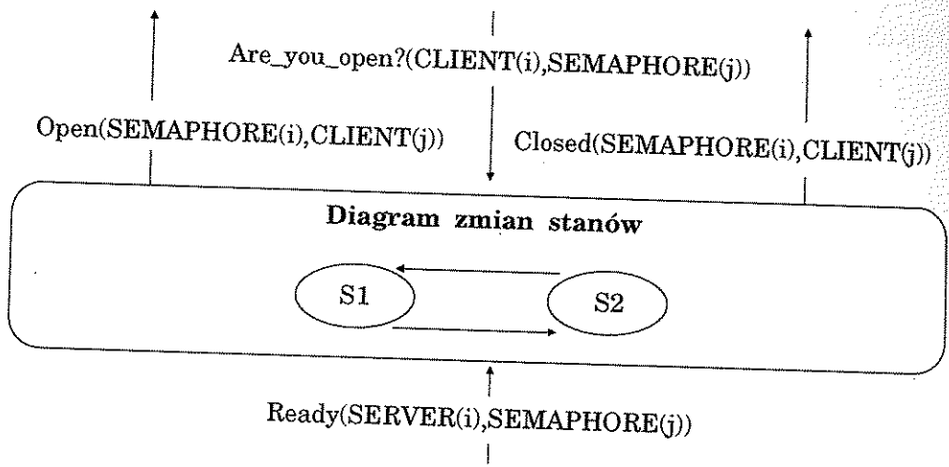
Napisz swoje rozwiązanie w Dokumentacji 8.

Dokumentacja 1

Typ obiektu: SEMAPHORE

Możliwe identyfikatory: {SEMAPHORE(1), SEMAPHORE(2), SEMAPHORE(3), ... }
Stan: {S1, S2}; S1 jest stanem początkowym.
Lista priorytetów: SERVER(1), SERVER(2), ... ,CLIENT(1),

Diagram komunikacji



Procedury przyjmowania

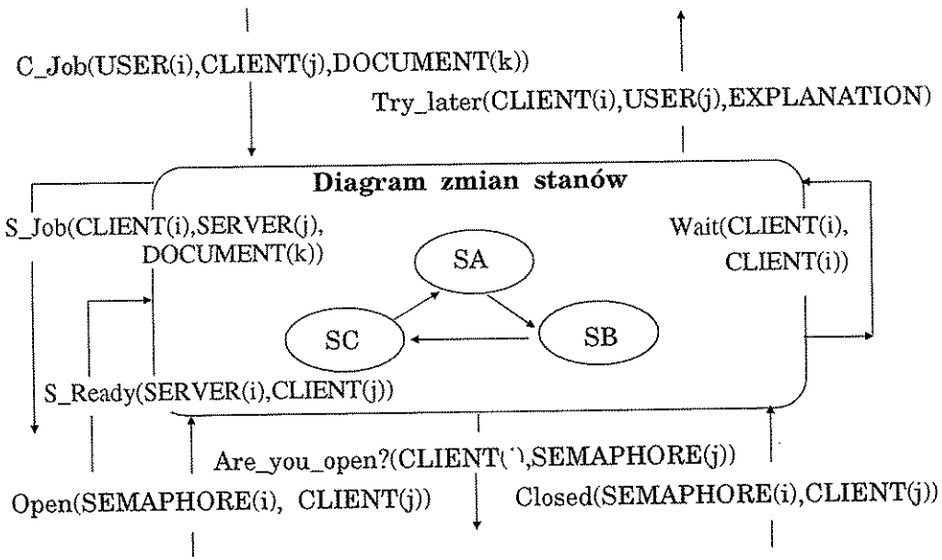
```
procedure Are_you_open?(Client,Semaphore)
begin
 if  State = S1
 then State ← S2
      Send("Open(Semaphore,Client)")
 else
 if   State = S2
 then Send("Closed(Semaphore,Client)")
end

procedure Ready(Server,Semaphore)
begin
 State ← S1
end
```

Dokumentacja 2

Typ obiektu: OBJECT

Możliwe identyfikatory: {CLIENT(1), CLIENT(2), CLIENT(3), ... }
Stan: {SA, SB, SC}; SA jest stanem początkowym.
Lista priorytetów: CLIENT, SERVER(1), SERVER(2), ..., SEMAPHORE(1), SEMAPHORE(2), ... , USER(1), USER(2), ...
Odliczanie: $\{ t \mid t \in N \}$, wartość początkowa: 0.
Waiting_Period: $\{ t \mid t \in N$ oraz $t > 0 \}$
Semaphore_Index: $\{ \hat{} i \mid i = 1, 2, ..., \text{Number\_of\_Servers} \}$
Number_of_Servers: $\{ i \mid i \in N$ oraz $i > 0 \}$

Diagram komunikacji

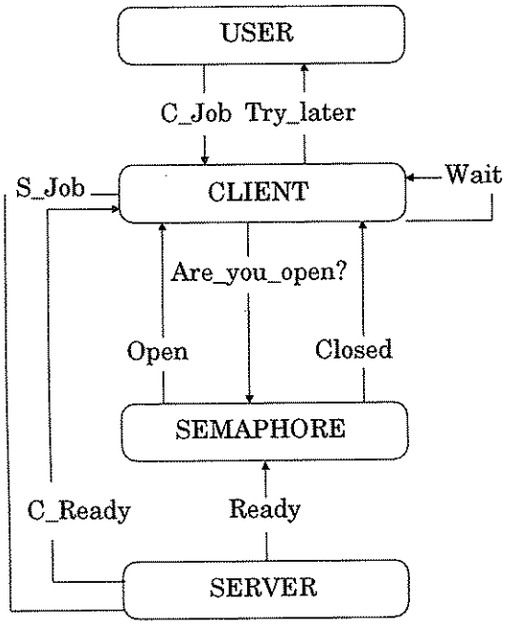


Procedury przyjmowania

```
procedure C_Job(User,Client,Document)
begin
 if   State = SA
 then State ← SB
      Send("Are_you_open?(Client,SEMAPHORE(Semaphore_Index))")
 else
 if   State ← SB
 then Send("Try_later(Client,User,Client_is_busy)")
 else
 if   State =SC
 then Send("Try_later(Client,User,Client_is_busy)")
end
procedure Open(Semaphore,Client)
begin
 if State = SB
 then State ← SC
      Send("S_Job(Client,Server,Document)")
end
procedure Closed(Semaphore,Client)
begin
 Countdown ← Waiting_Period
 Send("Wait(Client,Client)")
end
procedure Wait(Client,Client)
begin
 Countdown ← Countdown - 1
 if   Countdown > 0
 then Send("Wait(Client,Client)")
 else Send("Are_you_open?(Client,SEMAPHORE(Semaphore_Index))")
end
procedure C_Ready(Server,Client)
begin
 State ← SA
end
```

Dokumentacja 3

Schemat komunikacji



Lista Komunikatów ($i, j, k \in N$)

identyfikator	nadawca	odbiorca	zawartość
Are_you_open?	CLIENT(i)	SEMAPHORE(j)	–
C_Job	USER(i)	CLIENT(j)	DOCUMENT(k)
C_Ready	SERVER(i)	CLIENT(j)	–
Closed	SEMAPHORE(i)	CLIENT(j)	–
Open	SEMAPHORE(i)	CLIENT(j)	–
Ready	SERVER(i)	SEMAPHORE(j)	–
S_Job	CLIENT(i)	SERVER(j)	DOCUMENT(k)
Try_later	CLIENT(i)	USER(j)	EXPLANATION
Wait	CLIENT(i)	CLIENT(i)	–

Dokumentacja 4. Odpowiedź na podzadanie A.3

time	SEMAPHORE(1)		
	received messages	State/ Transition	sent messages
0		S1	
1		S1	
2	Are_you_open?	S1→S2	Open
3	Are_you_open?	S2	Closed
4		S2	
5	Ready	S2→S1	
6	Are_you_open? Are_you_open?	S1→S2 S2	Open Closed

time	CLIENT(1)			
	received messages	State/ Transition	Count-down	sent messages
0		SA	0	
1	C_Job	SA→SB	0	Are_you_open?
2		SB	0	
3	Open	SB→SC	0	S_Job
4		SC	0	
5	C_Ready C_Job	SC→SA SA→SB	0 0	Are_you_open?

time	CLIENT(2)			
	received messages	State/ Transition	Count-down	sent messages
0		SA	0	
1		SA	0	
2	C_Job	SA→SB	0	Are_you_open?
3		SB	0	
4	Closed	SB	0→1	Wait
5	Wait	SB	1→0	Are_you_open?

Dokumentacja 5

Ta dokumentacja zawierała pytania A.1 i A.2, w sformułowaniu z treści zadania, oraz wydzielone miejsca na udzielenie odpowiedzi.

Dokumentacja 6, Podzadanie B

Zmienione procedury przyjmowania 'Receive Procedures': C_Job oraz Wait

```
procedure C_Job(User,Client)
begin
 if   State = SA
 then State ← SB
      for Semaphore_Index←1 step 1 until Number_of_Servers
          Send("Are_you_open?(Client,
                              SEMAPHORE(Semaphore_Index))")
 else
 if   State = SB
 then Send("Try_later(Client,User,Client_is_busy)")
 else
 if   State = SC
 then Send("Try_later(Client,User,Client_is_busy)")
end
procedure Wait(Client,Client)
begin
 if   State = SB
 then Countdown ← Countdown - 1
      if   Countdown  >  0
      then Send("Wait(Client,Clie-t)")
      else
      if   Countdoun  <  0
      then Countdoun ← 0
      else for Semaphore_Index ← 1 step 1 until
               Number_of_Servers
               Send("Are_you_open?(Client,
                     SEMAPHORE(Semaphore_Index))")
end
```


W chwili 0 sytuacja w sieci LAN jest następująca:
Object: CLIENT(1), Client.State = SA, Waiting_Period = 2,
Number_of_Servers=2
Object: CLIENT(2), Client.State = SA, Waiting_Period = 1,
Number_of_Servers=2
Object: SEMAPHORE(1), Semaphore.State = S1
Object: SEMAPHORE(2), Semaphore.State = S1
W następujących chwilach, odbierane są następujące komunikaty:
W chwili 0: CLIENT(1) otrzymuje komunikat „C_Job" od użytkownika.
W chwili 0: CLIENT(2) otrzymuje komunikat „C_Job" od użytkownika.
W chwili 4: SEMAPHORE(1) otrzymuje komunikat „Ready".
Co się stanie w tej rozszerzonej sieci LAN zgodnie ze zmienioną definicją typu obiektów CLIENT?

Zaznacz właściwe odpowiedzi.

(a) Zadanie drukowania obiektu CLIENT(1) będzie obsłużone przez SERVER(1) i SERVER(2).
Zadanie drukowania obiektu CLIENT(2) nie będzie obsłużone.

(b) Zadanie drukowania obiektu CLIENT(1) będzie obsłużone raz przez SERVER(1).
Zadanie drukowania obiektu CLIENT(2) będzie obsłużone raz przez SERVER(2).

(c) Zadanie drukowania obiektu CLIENT(1) będzie obsłużone raz przez SERVER(1).
Zadanie drukowania obiektu CLIENT(2) będzie obsłużone raz przez SERVER(1).

(d) Zadanie drukowania obiektu CLIENT(1) będzie obsłużone raz przez SERVER(2).
Zadanie drukowania obiektu CLIENT(2) będzie obsłużone raz przez SERVER(2).

(e) Zadanie drukowania obiektu CLIENT(1) nie będzie obsłużone.
Zadanie drukowania obiektu CLIENT(2) będzie obsłużone przez SERVER(1) i SERVER(2).

Dokumentacja 7, Podzadanie C.1.

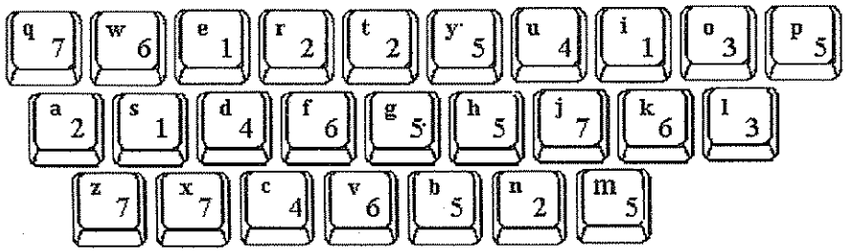
Czysta kartka w linie do umieszczenia na niej rozwiązania podzadania C.1.

Dokumentacja 8, Podzadanie C.2.

Czysta kartka w linie do umieszczenia na niej rozwiązania podzadania C.2.

9.2.5. Zadanie GRA LITEROWA

Gry literowe to popularne zabawy rodzinne, występują także w telewizyjnych teleturniejach. W jednej z wersji takiej gry każda litera ma pewną wartość i zbiera się litery tak, aby układając z nich jedno lub więcej słów uzyskać jak najwyższą możliwą liczbę punktów. Jeśli nie znasz „sztuki dobierania słów", będziesz próbował wszystkie wyrazy jakie znasz, czasem zaglądając do słownika, i liczyć później punkty. Oczywiście można to zrobić dokładniej posługując się komputerem.



Rysunek 1. Wszystkie małe litery i ich wartości.

Mając wartości punktowe podane na Rysunku 1, listę angielskich słów oraz wybrane litery, znajdź słowa lub pary słów, które można ułożyć, dające najwyższą możliwą liczbę punktów.

Dane wejściowe

Plik wejściowy INPUT.TXT zawiera jeden wiersz z napisem zbudowanym z małych liter (od „a" do „z"): wybrane litery. Ten napis składa się z co najmniej 3 i co najwyżej 7 liter w dowolnym porządku. Żadna litera nie może występować w wierszu pliku wyjściowego więcej razy niż w wierszu pliku INPUT.TXT.

Plik słownikowy WORDS.TXT zawiera co najwyżej 40.000 wierszy. Na końcu tego pliku jest wiersz z pojedynczą kropką („."). Każdy z pozostałych wierszy zawiera napis złożony z co najmniej 3 i co najwyżej 7 małych liter. Słowa w tym pliku są posortowane alfabetycznie. Plik WORDS.TXT nie zawiera powtórzeń wyrazów.

Dane wyjściowe

W pierwszym wierszu pliku OUTPUT.TXT twój program powinien napisać najwyższą możliwą do uzyskania liczbę punktów (Podzadanie A), a w następnych wierszach wszystkie słowa lub pary słów z pliku WORDS.TXT, mające taką wartość (Podzadanie B). Użyj wartości liter podanych na rys. 1.

Gdy z wybranych liter można utworzyć parę słów, to te słowa powinny być wypisane w tym samym wierszu i oddzielone odstępem. Nie powtarzaj par: na przykład „rag prom" i „prom rag" tworzą tę samą parę, wobec tego należy wpisać tylko jedną z nich. Para słów w wierszu pliku wyjściowego może składać się z dwóch identycznych słów.

Przykład danych wejściowych i wyjściowych

WORDS.TXT
profile
program
prom
rag
ram
rom

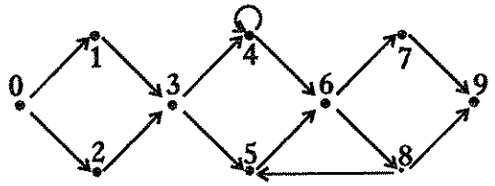
INPUT.TXT
prmgroa
OUTPUT.TXT
24
program
prom rag

9.2.6. Zadanie WYŚCIG ULICZNY

Rysunek 1 przedstawia przykład trasy ulicznego wyścigu. Są tam punkty, ponumerowane od 0 do N (tutaj $N = 9$) i strzałki łączące te punkty. Punkt 0 jest startem wyścigu, a punkt N – metą. Strzałki reprezentują ulice jednokierunkowe. Uczestnicy wyścigu poruszają się od punktu do punktu ulicami jedynie w kierunku strzałek. W każdym punkcie uczestnik może wybrać dowolną z wychodzących strzałek.

Poprawnie wytyczona trasa ma następujące właściwości:

1. Ze startu można dotrzeć do każdego punktu na trasie.
2. Z każdego punktu na trasie można dotrzeć do mety.
3. Żadna strzałka nie wychodzi z mety.



Rysunek 1. Trasa ulicznego wyścigu z 10 punktami.

Uczestnik wyścigu nie musi odwiedzić każdego punktu na trasie, aby dotrzeć do mety. Jednakże niektóre punkty sa nieomijalne. W naszym przykładzie są to punkty 0, 3, 6, 9. Mając zadaną poprawnie wytyczoną trasę, twój program powinien okreslić zbiór nieomijalnych punktów, które wszyscy uczestnicy muszą odwiedzić, za wyjątkiem startu i mety (Podzadanie A).

Załóżmy, że wyścig ma się odbywać w ciągu dwóch kolejnych dni. W tym celu, trzeba rozłożyć (rozciąć) trasę na dwie części, po jednej na każdy dzień. Pierwszego dnia startem jest punkt 0, a metą jest pewien punkt rozcięcia. Drugiego dnia startem jest ten punkt rozcięcia, zaś meta znajduje się w punkcie N. Mając zadaną poprawnie wytyczoną trasę, twój program powinien określić zbiór punktów rozcięcia (Podzadanie B).

Punkt S jest **punktem rozcięcia** poprawnie wytyczonej trasy C, jeśli S nie jest startem ani metą C, oraz trasa może być rozcięta na dwie poprawnie wytyczone trasy, które nie mają wspólnych strzałek i S jest jedynym wspólnym punktem tych tras. W naszym przykładzie jedynie punkt 3 jest punktem rozcięcia.

Dane wejściowe

Plik wejściowy INPUT.TXT opisuje pewną poprawnie wytyczoną trasę z co najwyżej 50 punktami i 100 strzałkami. W tym pliku jest $N+1$ wierszy. Początkowe N wierszy zawiera końce strzałek zaczynających się odpowiednio w punktach od 0 do $N-1$. Każdy z tych wierszy jest zakończony liczbą –2. Ostatni wiersz zawiera liczbę –1.

Dane wyjściowe

Twój program powinien wpisać dwa wiersze do pliku OUTPUT.TXT. Pierwszy wiersz powinien zawierać liczbę nieomijalnych punktów na trasie opisanej w pliku wejściowym, a następnie numery tych punktów w dowolnym porządku (Podzadanie A). Drugi wiersz powinien zawierać liczbę punktów rozcięcia tej trasy, a następnie listę tych punktów w dowolnym porządku (Podzadanie B).

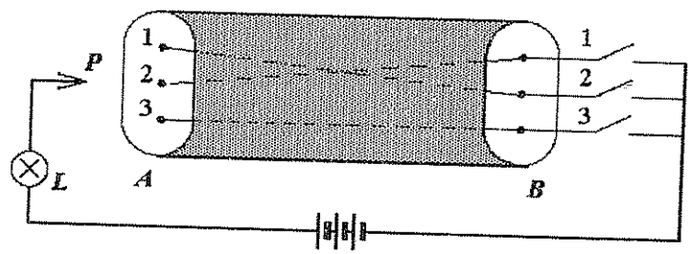
Przykład danych wejściowych i wyjściowych

Dla przykładu z rysunku 1 zawartości plików wejściowego i wyjściowego są następujące:

INPUT.TXT
```
1 2 –2
3 –2
3 –2
5 4 –2
6 4 –2
6 –2
7 8 –2
9 –2
5 9 –2
–1
```
OUTPUT.TXT
```
2 3 6
1 3
```

9.2.7. Zadanie PRZEWODY I WŁĄCZNIKI

Na rysunku 1 kabel zawierający trzy przewody łączy stronę A ze stroną B. Po stronie A trzy przewody mają numery 1, 2, 3. Po stronie B przewody 1 i 3 są przyłączone do włącznika 3, zaś przewód 2 jest przyłączony do włącznika 1.



Rysunek 1. Kabel zawierający trzy przewody i trzy włączniki.

W ogólnym przypadku kabel zawiera m przewodów ($1 \le m \le 90$), ponumerowanych od 1 do m po stronie A, a po stronie B znajduje się m włączników, ponumerowanych od 1 do m. Każdy przewód jest przyłączony do dokładnie jednego z tych włączników. Do każdego włącznika może być przyłączone zero lub więcej przewodów.

Pomiary

Twój program powinien określić za pomocą pewnych pomiarów, jak przewody są połączone z włącznikami. Każdy włącznik może być włączony lub rozłączony. Na początku wszystkie włączniki są rozłączone. Przewód może być badany po stronie *A* za pomocą próbnika *P*: Żarówka *L* zaświeci się wtedy i tylko wtedy gdy przewód badany próbnikiem jest przyłączony do włącznika, który jest włączony.

Twój program zaczyna od odczytania ze *standardowego wejścia* jednego wiersza z liczbą *m*. Później twój program może wydawać trzy rodzaje poleceń wypisując je na *standardowym wyjściu* programu. Każde polecenie zaczyna się od pojedynczej wielkiej litery: T (zbadaj przewód), C (zmień stan włącznika), D (koniec pracy). Po poleceniu T następuje numer przewodu, po C – numer włącznika, zaś po D – następuje lista, której *i*-tym elementem jest numer włącznika, do którego jest podłączony *i*-ty przewód.

Po wydaniu polecenia T lub C, twój program powinien odczytać jeden wiersz ze *standardowego wejścia*. Reakcją na polecenie T jest Y (Yes), gdy ten przewód jest przyłączony do włączonego włącznika (żarówka zaświeciła się), lub N (No) w przeciwnym przypadku. Reakcją na polecenie C jest Y (Yes), gdy po zmianie stanu włącznika jest on włączony, lub N (No) w przeciwnym przypadku. Polecenie C zmienia stan włącznika (jeśli był włączony, będzie po tym rozłączony, i odwrotnie); reakcje podaje się tylko dla większej czytelności.

Twój program może wydawać polecenia T i C w dowolnej kolejności. Na końcu wydaje polecenie D i kończy swoje działanie. Całkowita liczba poleceń wydanych przez twój program nie powinna przekroczyć dziewięciuset (900).

Przykład

Przykładowa konwersacja dotyczaca rys. 1 może mieć następującą postać:

Standardowe wyjście	*Standardowe wejście*
	3
C 3	Y
T 1	Y
T 2	N
T 3	Y
C 3	N
C 2	Y
T 2	N
D 3 1 3	

Uwaga: Aby zapewnić prawidłowe użycie standardowego wejścia i wyjścia, nie używaj w języku Pascal pakietu (unit) `Crt`.

LITERATURA

Poniżej zamieszczamy listę książek w języku polskim, które są poświęcone algorytmom i ich kompletnym realizacjom komputerowym. Do wielu z tych książek odwołujemy się w rozwiązaniach zadań, mogą być również przydatne w przygotowywaniu się do zawodów następnych olimpiad.

1. *I Olimpiada Informatyczna 1993/1994*, Warszawa, Wrocław 1994* (w tych materiałach powołujemy się na tę pozycję pisząc *I-OI*).
2. Aho, A.V., Hopcroft, J.E., Ullman, J.D., *Projektowanie i analiza algorytmów komputerowych*, PWN, Warszawa 1983.
3. Banachowski, L., Kreczmar, A., *Elementy analizy algorytmów*, WNT, Warszawa 1982.
4. Banachowski, L., Kreczmar, A., Rytter, W., *Analiza algorytmów i struktur danych*, WNT, Warszawa 1987.
5. Bentley, J., *Perełki oprogramowania*, WNT, Warszawa 1992.
6. Graham, R., Knuth, D.E., Patashnik, O., *Matematyka konkretna*, Wydawnictwo Naukowe PWN (w przygotowaniu).
7. Harel, D., *Algorytmika – Rzecz o istocie informatyki*, WNT, Warszawa 1992.
8. Lipski, W., *Kombinatoryka dla programistów*, WNT, Warszawa 1989.
9. Reingold, E.M., Nievergelt, J., Deo, N., *Algorytmy kombinatoryczne*, WNT, Warszawa 1985.
10. Sysło, M.M., Deo, N., Kowalik, J.S., *Algorytmy optymalizacji dyskretnej z programami w języku Pascal*, Wydawnictwo Naukowe PWN, Warszawa 1993.
11. Wirth, N., *Algorytmy + struktry danych = programy*, WNT, Warszawa 1989.

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