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

Testy

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

Zadaniem trzech testów było sprawdzenie szybkości działania ocenianych programów.

7.2. Zawody II stopnia

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

Treść zadania PIĘĆ MONET

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

Zadanie

Rozwiązanie możesz zapisać w Pascalu lub w języku C.

Pascal

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

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

Kod źródłowy programu jest zapisany w następujących plikach:

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

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

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

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

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

Rozwiązanie zapisz w pliku USORT.PAS:

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

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

Język C

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

Kod źródłowy programu jest zapisany w następujących plikach:

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

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

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

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

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

Rozwiązanie zapisz w pliku USORT.C:

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

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

Rozwiązanie zadania PIĘĆ MONET

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

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

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

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

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

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

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

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

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

Faza druga. Porównujemy C z E.

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

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

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

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

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

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

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

Omówienie rozwiązań podanych przez uczniów

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

Testy

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

7.2.2. Zadanie WIEŻE (Autor: Krzysztof Diks)

Treść zadania WIEŻE

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

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