informatyka+

Wszechnica Poranna: Algorytmika i programowanie

Wprowadzenie do algorytmiki i programowania – wyszukiwanie i porządkowanie informacji

Maciej M Sysło

Człowiek – najlepsza inwestycja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Rodzaj zajęć: Wszechnica Poranna
Tytuł: Wprowadzenie do algorytmiki i programowania
— wyszukiwanie i porządkowanie informacji
Autor: prof. dr hab. Maciej M Sysło
Redaktor merytoryczny: prof. dr hab. Maciej M Sysło

Zeszyt dydaktyczny opracowany w ramach projektu edukacyjnego **Informatyka+**
— ponadregionalny program rozwijania kompetencji uczniów szkół ponadgimnazjalnych
w zakresie technologii informacyjno-komunikacyjnych (ICT).
www.informatykaplus.edu.pl
kontakt@informatykaplus.edu.pl

Wydawca: Warszawska Wyższa Szkoła Informatyki
ul. Lewartowskiego 17, 00-169 Warszawa
www.wwsi.edu.pl
rektorat@wwsi.edu.pl

Projekt graficzny: FRYCZ I WICHA

Warszawa 2009

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska
Wyższa Szkoła
Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Wyszukiwanie i porządkowanie informacji

Maciej M. Sysło
Uniwersytet Wrocławski, UMK w Toruniu
syslo@ii.uni.wroc.pl, syslo@mat.uni.torun.pl

Streszczenie

Te zajęcia są wprowadzeniem do algorytmiki i programowania. Na przykładach bardzo prostych problemów, takich jak: znajdowanie największego i/lub najmniejszego elementu w ciągu, wyłanianie zwycięzcy i drugiego zawodnika w turnieju, porządkowanie ciągu liczby oraz poszukiwanie elementów w zbiorach nieuporządkowanych i uporządkowanych, przedstawione jest podejście do rozwiązywania problemów w postaci algorytmów i do ich komputerowej implementacji w języku Pascal lub C++. Omawiane są m.in. specyfikacja problemu, schematy blokowe algorytmów, podstawowe struktury danych (ciąg i tablica) oraz pracochłonność (złożoność) algorytmów.

Na warsztatach zostają wprowadzone podstawowe instrukcje języka programowania (iteracyjna i warunkowa oraz procedura i funkcja niestandardowa), wystarczające do zaprogramowania i uruchomienia komputerowych realizacji algorytmów omówionych na wykładzie. Wykorzystywane jest oprogramowanie edukacyjne, ułatwiające zrozumienie działania algorytmów i umożliwiające wykonywanie eksperymentów z algorytmami bez konieczności ich programowania. Przytoczono ciekawe przykłady zastosowań omawianych zagadnień.

Rozważania są prowadzone na elementarnym poziomie i do ich wysłuchania oraz wzięcia udziału w warsztatach wystarczy znajomość informatyki wyniesiona z gimnazjum. Te zajęcia są adresowane do wszystkich uczniów w szkołach ponadgimnazjalnych, zgodnie bowiem z nową podstawą programową, kształceniem umiejętności algorytmicznego rozwiązywania problemów mają być objęci wszyscy uczniowie.

Spis treści

1 ALGORYTM, ALGORYTMIKA I ALGORYTMICZNE ROZWIĄZYWANIE PROBLEMÓW

Ten rozdział jest krótkim wprowadzeniem do zajęć w module „Algorytmika i programowanie". Krótko wyjaśniamy w nim podstawowe pojęcia oraz stosowane na zajęciach podejście do rozwiązywania problemów z pomocą komputera.

Algorytm

Powszechnie przyjmuje się, że **algorytm** jest opisem krok po kroku rozwiązania postawionego problemu lub sposobu osiągnięcia jakiegoś celu. To pojęcie wywodzi się z matematyki i informatyki – za pierwszy algorytm uznaje się bowiem algorytm Euklidesa, podany ponad 2300 lat temu. W ostatnich latach algorytm stał się bardzo popularnym synonimem przepisu lub instrukcji postępowania.

W szkole, algorytm pojawia się po raz pierwszy na lekcjach matematyki już w szkole podstawowej, na przykład jako algorytm pisemnego dodawania dwóch liczb, wiele klas wcześniej, zanim staje się przedmiotem zajęć informatycznych.

O znaczeniu algorytmów w informatyce może świadczyć następujące określenie, przyjmowane za definicję informatyki:

informatyka jest dziedziną wiedzy i działalności zajmującą się algorytmami

W tej definicji informatyki nie ma dużej przesady, gdyż zawarte są w niej pośrednio inne pojęcia stosowane do definiowania informatyki: **komputery** – jako urządzenia wykonujące odpowiednio dla nich zapisane algorytmy (czyli niejako wprawiane w ruch algorytmami); **informacja** – jako materiał przetwarzany i produkowany przez komputery; **programowanie** – jako zespół metod i środków (np. języków i systemów użytkowych) do zapisywania algorytmów w postaci programów.

Położenie nacisku w poznawaniu informatyki na algorytmy jest jeszcze uzasadnione tym, że zarówno konstrukcje komputerów, jak i ich oprogramowanie bardzo szybko się starzeją, natomiast podstawy stosowania komputerów, które są przedmiotem zainteresowań algorytmiki, zmieniają się bardzo powoli, a niektóre z nich w ogóle nie ulegają zmianie.

Algorytmy, zwłaszcza w swoim popularnym znaczeniu, występują wszędzie wokół nas – niemal każdy ruch człowieka, zarówno angażujący jego mięśnie, jak i będący jedynie działaniem umysłu, jest wykonywany według jakiegoś przepisu postępowania, którego nie zawsze jesteśmy nawet świadomi. Wiele naszych czynności potrafimy wyabstrahować i podać w postaci precyzyjnego opisu, ale w bardzo wielu przypadkach nie potrafimy nawet powtórzyć, jak to się dzieje lub jak to się stało[1].

Nie wszystkie postępowania z naszego otoczenia, nazywane algorytmami, są ściśle związane z komputerami i nie wszystkie przepisy działań można uznać za algorytmy w znaczeniu informatycznym. Na przykład nie są nimi na ogół przepisy kulinarne, chociaż odwołuje się do nich David Harel w swoim fundamentalnym dziele o algorytmach i algorytmice [5]. Otóż przepis np. na sporządzenie „ciągutki z wiśniami", którą zachwycała się Alicja w Krainie Czarów, nie jest algorytmem, gdyż nie ma dwóch osób, które na jego podstawie, dysponując tymi samymi produktami, zrobiłyby taką samą, czyli jednakowo smakującą ciągutkę. Nie może być bowiem algorytmem przepis, który dla identycznych danych daje różne wyniki w dwóch różnych wykonaniach, jak to najczęściej bywa w przypadku robienia potraw według „algorytmów kulinarnych".

Algorytmika

Algorytmika to dział informatyki, zajmujący się różnymi aspektami tworzenia i analizowania algorytmów, przede wszystkim w odniesieniu do ich roli jako precyzyjnego opisu postępowania, mającego na celu znalezienie rozwiązania postawionego problemu. Algorytm może być wykonywany przez człowieka, przez komputer lub w inny sposób, np. przez specjalnie dla niego zbudowane urządzenie. W ostatnich latach postęp w rozwoju komputerów i informatyki był nierozerwalnie związany z rozwojem coraz doskonalszych algorytmów.

Informatyka jest dziedziną zajmującą się rozwiązywaniem problemów z wykorzystaniem komputerów. O znaczeniu algorytmów w informatyce może świadczyć fakt, że każdy program komputerowy działa zgodnie z jakimś algorytmem, a więc zanim zadamy komputerowi nowe zadanie do wykonania powinniśmy umieć „wytłumaczyć" mu dokładnie, co ma robić. Bardzo trafnie to sformułował Donald E. Knuth, jeden z najznakomitszych, żyjących informatyków:

[1] Interesująco ujął to J. Nievergelt w artykule [7] – Jest tak, jakby na przykład stonoga chciała wyjaśnić, w jakiej kolejności wprawia w ruch swoje nogi, ale z przerażeniem stwierdza, że nie może iść dalej.

Mówi się często, że człowiek dotąd nie zrozumie czegoś,
zanim nie nauczy tego – kogoś innego.
W rzeczywistości,
człowiek nie zrozumie czegoś naprawdę,
zanim nie zdoła nauczyć tego – komputera.

Staramy się, by prezentowane algorytmy były jak najprostsze i by działały jak najszybciej. To ostatnie żądanie może wydawać się dziwne, przecież dysponujemy już teraz bardzo szybkimi komputerami i szybkość działania procesorów stale rośnie (według prawa Moore'a podwaja się co 18 miesięcy). Mimo to istnieją problemy, których obecnie nie jest w stanie rozwiązać żaden komputer i zwiększenie szybkości komputerów niewiele pomoże, kluczowe więc staje się opracowywanie coraz szybszych algorytmów. Jak to ujął Ralf Gomory, szef ośrodka badawczego IBM:

Najlepszym sposobem przyspieszania komputerów
jest obarczanie ich mniejszą liczbą działań.

Algorytmiczne rozwiązywanie problemów

Komputer jest stosowany do rozwiązywania problemów zarówno przez profesjonalnych informatyków, którzy projektują i tworzą oprogramowanie, jak i przez tych, którzy stosują tylko technologię informacyjno-komunikacyjną, czyli nie wykraczają poza posługiwanie się gotowymi narzędziami informatycznymi. W obu przypadkach ma zastosowanie podejście do **rozwiązywania problemów algorytmicznych**, która polega na systematycznej pracy nad komputerowym rozwiązaniem problemu i obejmuje cały proces projektowania i otrzymania rozwiązania. Celem nadrzędnym tej metodologii jest otrzymanie **dobrego rozwiązania**, czyli takiego, które jest:

- **zrozumiałe dla każdego,** kto zna dziedzinę rozwiązywanego problemu i użyte narzędzia komputerowe,
- **poprawne,** czyli spełnia specyfikację problemu, a więc dokładny opis problemu,
- **efektywne,** czyli niepotrzebnie nie marnuje zasobów komputerowych, czasu i pamięci.

Ta metoda składa się z następujących sześciu etapów:

1. **Opis i analiza sytuacji problemowej.** Na podstawie opisu i analizy sytuacji problemowej należy w pełni zrozumieć, na czym polega problem, jakie są dane dla problemu i jakich oczekujemy wyników, oraz jakie są możliwe ograniczenia.

2. **Sporządzenie specyfikacji problemu,** czyli dokładnego opisu problemu na podstawie rezultatów etapu 1. **Specyfikacja problemu** zawiera:
   - opis danych,
   - opis wyników,
   - opis relacji (powiązań, zależności) między danymi i wynikami.

   Specyfikacja jest wykorzystana w następnym etapie jako specyfikacja tworzonego rozwiązania (np. programu).

3. **Zaprojektowanie rozwiązania.** Dla sporządzonej na poprzednim etapie specyfikacji problemu, jest projektowane rozwiązanie komputerowe (np. program), czyli wybierany odpowiedni algorytm i dobierane do niego struktury danych. Wybierane jest także środowisko komputerowe (np. język programowania), w którym będzie realizowane rozwiązanie na komputerze.

4. **Komputerowa realizacja rozwiązania.** Dla projektu rozwiązania, opracowanego na poprzednim etapie, jest budowane kompletne rozwiązanie komputerowe, np. w postaci programu w wybranym języku programowania. Następnie, testowana jest poprawność rozwiązania komputerowego i badana jego efektywność działania na różnych danych.

5. **Testowanie rozwiązania.** Ten etap jest poświęcony na systematyczną weryfikację poprawności rozwiązania i testowanie jego własności, w tym zgodności ze specyfikacją.

6. **Prezentacja rozwiązania.** Dla otrzymanego rozwiązania należy jeszcze opracować dokumentację i pomoc dla (innego) użytkownika. Cały proces rozwiązywania problemu kończy prezentacja innym zainteresowanym osobom (uczniom, nauczycielowi) sposobu otrzymania rozwiązania oraz samego rozwiązania wraz z dokumentacją.

Chociaż powyższa metodologia jest stosowana głównie do otrzymywania komputerowych rozwiązań, które mają postać programów napisanych w wybranym języku programowania, może być zastosowana również do otrzymywania rozwiązań komputerowych większości problemów z obszaru zastosowań informatyki i posługiwania się technologią informacyjno-komunikacyjną[2], czyli gotowym oprogramowaniem.

Dwie uwagi do powyższych rozważań.

Uwaga 1. Wszyscy, w mniejszym lub większym stopniu, zmagamy się z problemami, pochodzącymi z różnych dziedzin (przedmiotów). W naszych rozważaniach, problem nie jest jednak wyzwaniem nie do pokonania, przyjmujemy bowiem, że **problem** jest sytuacją, w której uczeń ma przedstawić jej rozwiązanie bazując na tym, co wie, ale nie ma powiedziane, jak to ma zrobić. Problem na ogół zawiera pewną trudność, nie jest rutynowym zadaniem. Na takie sytuacje problemowe rozszerzamy pojęcie problemu, wymagającego przedstawienia rozwiązania komputerowego.

Uwaga 2. W tych rozważaniach rozszerzamy także pojęcie **programowania**. Jak powszechnie wiadomo, komputery wykonują tylko programy. Użytkownik komputera może korzystać z istniejących programów (np. za pakietu Office), a może także posługiwać się własnymi programami, napisanymi w języku programowania, który „rozumieją" komputery. W szkole nie ma zbyt wiele czasu, by uczyć programowania, uczniowie też nie są odpowiednio przygotowani do programowania komputerów. Istnieje jednak wiele sposobności, by kształcić zdolność komunikowania się z komputerem za pomocą programów, które powstają w inny sposób niż za pomocą programowania w wybranym języku programowania. Szczególnym przypadkiem takich programów jest oprogramowanie edukacyjne, które służy do wykonywania i śledzenia działania algorytmów. „Programowanie" w przypadku takiego oprogramowania polega na dobieraniu odpowiednich parametrów, które mają wpływ na działanie algorytmów i tym samym umożliwiają lepsze zapoznanie się z nimi.

[2] W najnowszej podstawie programowej dla przedmiotu informatyka, przyjętej do realizacji pod koniec 2008 roku, to podejście jest zalecane jako podstawowa metodologia rozwiązywania problemów z pomocą komputera.

2 PIERWSZY ALGORYTM – PRZESZUKIWANIE ZBIORU

Zaczniemy nasze rozważania od bardzo prostego problemu, który każdy z Was, jak i każdy człowiek, rozwiązuje wielokrotnie w ciągu dnia. Chodzi o znajdowanie w zbiorze elementu, który ma określoną własność. Oto przykładowe sytuacje problemowe.

Ćwiczenie 1. Opisz, na czym polega każdy z opisanych niżej problemów, wymieniając dane i poszukiwany wynik. Zaproponuj sposób znajdowania poszukiwanego elementu:
- znajdź najwyższego ucznia w swojej klasie; a jak zmieni się Twój algorytm, jeśli chciałbyś znaleźć w klasie najniższego ucznia?
- znajdź w swojej klasie ucznia, któremu droga do szkoły zabiera najwięcej czasu;
- znajdź najstarszego ucznia w swojej szkole; jak zmieni się Twój algorytm, gdybyś chciał znaleźć w szkole najmłodszego ucznia?
- znajdź największą kartę w potasowanej talii kart;
- znajdź najlepszego gracza w warcaby w swojej klasie (zakładamy, że wszyscy potrafią grać w warcaby);
- znajdź najlepszego tenisistę w swojej klasie.

Tego typu problemy pojawiają się bardzo często, również w obliczeniach komputerowych, i na ogół są rozwiązywane w dość naturalny sposób – przeglądany jest cały zbiór, by znaleźć poszukiwany element. Zastanowimy się, jak dobra jest to metoda, i czy może istnieje szybsza metoda znajdowania w zbiorze elementu o określonych własnościach.

Postawiony problem może wydać się zbyt prosty, by zajmować się nim na informatyce – każdy uczeń zapewne potrafi wskazać metodę rozwiązywania, polegającą na systematycznym **przeszukanie** całego zbioru danych. Tak pojawia się metoda przeszukiwania ciągu, którą można nazwać przeszukiwaniem **liniowym**. Przy tej okazji w dyskusji powinna pojawić się również **metoda pucharowa**, która jest często stosowana, gdy porównania muszą być wykonane w bezpośrednim „spotkaniu" elementów. Poza tym ta metoda umożliwia intuicyjne wprowadzenie **drzewa algorytmów**, które może służyć do wyjaśnienia wielu innych kwestii, związanych głównie ze złożonością algorytmów. po pierwsze – na jego przykładzie zilustrujemy wiele różnych aspektów związanych z problemami i ich rozwiązywaniem w postaci al-

gorytmicznej oraz wykorzystaniem do tego celu komputerów, a po drugie – wskażemy na praktyczne znaczenie rozważanego problemu.

Założenia

Poczyńmy najpierw pewne założenia.

Założenie 1. Na początku wykluczamy, że przeszukiwane zbiory elementów są uporządkowane, np. klasa – od najwyższego do najniższego ucznia lub odwrotnie, szkoła – od najmłodszego do najstarszego ucznia lub odwrotnie, Gdyby tak było, to rozwiązanie problemów z ćwicz. 1 i im podobnych byłoby dziecinnie łatwe – wystarczyłoby wziąć element z początku albo z końca takiego uporządkowania.

Założenie 2. Przyjmujemy także, że nie interesują nas algorytmy rozwiązywania przedstawionych sytuacji problemowych, które w pierwszym kroku porządkują zbiór przeszukiwany, a następnie już prosto znajdują poszukiwane elementy – to założenie wynika z dalszych rozważań.

W dalszej części zajmiemy się sytuacjami, w których przeszukiwane zbiory są uporządkowane (rozdz. 6).

Z powyższych założeń wynika dość naturalny wniosek, że aby znaleźć w zbiorze poszukiwany element musimy przejrzeć wszystkie elementy zbioru, gdyż jakikolwiek pominięty element mógłby okazać się tym szukanym elementem.

Przy projektowaniu algorytmów istotne jest również określenie, jakie działania (operacje) mogą być wykonywane w algorytmie. W przypadku problemu poszukiwania szczególnego elementu w zbiorze wystarczy, jeśli będziemy umieli porównać elementy między sobą. Co więcej, w większości problemów w ćwicz. 1, porównanie elementów sprowadza się do porównania liczb, właściwych dla porównywanych elementów, a oznaczających: wzrost, czas na dojście do szkoły (np. liczony w minutach), wiek. Elementy zbiorów utożsamiamy więc z ich wartościami i wartości te nazywamy **danymi**, chociaż często prowadzimy rozważania w języku problemu, posługując się nazwami elementów: wzrost, wiek itp.

Przy porównywaniu kart należy uwzględnić ich kolory i wartości. Natomiast, by znaleźć w klasie najlepszego gracza w tenisa, należy zorganizować turniej – ten problem omówimy w dalszej części zajęć (patrz rozdz. 3).

Specyfikacja problemu

Dla uproszczenia rozważań można więc założyć, że dany jest pewien zbiór liczb A i w tym zbiorze należy znaleźć liczbę najmniejszą (lub największą). Przyjmijmy, że tych liczb jest n i oznaczmy je jako ciąg liczb: $x_1, x_2, \ldots, x_n$. Możemy teraz podać **specyfikację** rozważanego problemu:

Problem Min – Znajdowanie najmniejszego elementu w zbiorze

Dane: Liczba naturalna n i zbiór n liczb, dany w postaci ciągu $x_1, x_2, \ldots, x_n$.
Wynik: Najmniejsza spośród liczb $x_1, x_2, \ldots, x_n$ – oznaczmy jej wartość przez *min*.

Algorytm Min

Dla powyższej specyfikacji podamy teraz algorytm, który polega na przejrzeniu ciągu danych od początku do końca. Opis algorytmu poprzedza specyfikacją problemu, który ten algorytm rozwiązuje – tak będziemy na ogół postępować w każdym przypadku. Przedstawiony poniżej opis algorytmu ma postać **listy kroków**. O innych sposobach przedstawiania algorytmów piszemy w dalszej części.

Algorytm Min – znajdowanie najmniejszego elementu w zbiorze

Dane: Liczba naturalna n i zbiór n liczb, dany w postaci ciągu $x_1, x_2, \ldots, x_n$.
Wynik: Najmniejsza spośród liczb $x_1, x_2, \ldots, x_n$ – oznaczmy jej wartość przez *min*.
Krok 1. Przyjmij za *min* pierwszy element w zbiorze (w ciągu),
czyli przypisz $min := x_1$.
Krok 2. Dla kolejnych elementów x_i, gdzie $i = 2, 3, \ldots, n$,
jeśli *min* jest mniejsze niż x_i, to za *min* przyjmij x_i,
czyli, jeśli $min < x_i$, to przypisz $min := x_i$.

Uwaga. W opisie algorytmu pojawiło się polecenie (instrukcja) **przypisania**[3], np. $min := x_1$, w której występuje symbol :=, złożony z dwóch znaków: dwukropka i równości. Przypisanie oznacza nadanie wielkości (zmiennej) stojącej po lewej strony tego symbolu wartości równej wartości wyrażenia (w szczególnym przypadku to wyrażenie może być zmienną) znajdującego się po prawej stronie

[3] Polecenie przypisania jest czasem nazywane niepoprawnie podstawieniem.

tego symbolu. Przypisanie jest stosowane na przykład wtedy, gdy należy zmienić wartość zmiennej, np. $i := i + 1$ – w tym przypadku ta sama zmienna występuje po lewej i po prawej stronie symbolu przypisania. Polecenie przypisania występuje w większości języków programowania, stosowane są tylko różne symbole i ich uproszczenia dla jego oznaczenia. W schemacie blokowym na rys. 2 symbolem przypisania jest strzałka ← .

Metoda zastosowana w algorytmie **Min**, polegająca na badaniu elementów ciągu danych w kolejności, w jakiej są ustawione, nazywa się **przeszukiwaniem liniowym**, w odróżnieniu od przeszukiwania przez połowienie (lub binarnego), o którym jest mowa w rozdz. 6.

Demonstracja działania algorytmu Min

Działanie algorytmu **Min** można zademonstrować posługując się programem edukacyjnym **Maszyna sortująca** (patrz rys. 1), który jest udostępniony wraz z tymi materiałami. W tym programie można ustalić liczbę elementów w ciągu (między 1 i 16) i wybrać rodzaj ciągu danych, który może zawierać elementy: losowe, posortowane rosnąco lub posortowane malejąco. Ponieważ ten program służy do porządkowania ciągu, o czym będzie mowa w dalszej części zajęć (patrz rozdz. 5), zalecamy tutaj wykonanie demonstracji pracą krokową i przerwanie jej po znalezieniu najmniejszego elementu w ciągu – następuje to w momencie, gdy dolna zielona strzałka znajdzie się pod ostatnim elementem w ciągu i wygaszony zostanie czerwony kolor, wyróżniający porównywane elementy.

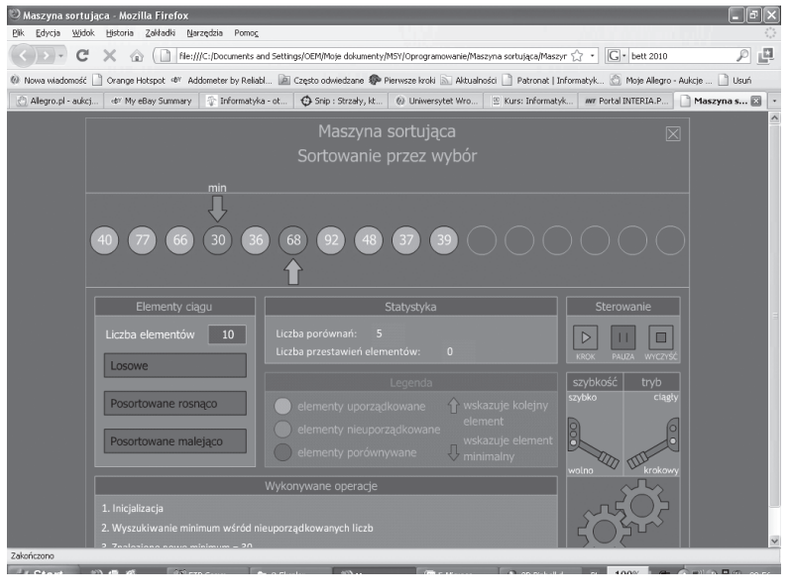
Ćwiczenie 2. Posłuż się programem **Maszyna sortująca** w opisany wyżej sposób, by obejrzeć działanie algorytmu **Min**. Wybieraj ciągi różnej długości, złożone z elementów losowych, uporządkowanych i odwrotnie uporządkowanych.

Algorytm Max – Prosta modyfikacja algorytmu Min

Podany powyżej algorytm znajdowania najmniejszej liczby w ciągu danych może być łatwo zmodyfikowany do znajdowania największego elementu w ciągu. Pozostawiamy to jako proste ćwiczenie.

Ćwiczenie 3. Korzystając z algorytmu **Min** zapisz algorytm **Max**, który znajduje największy element w ciągu danych.

Ćwiczenie 4. Następnie zmodyfikuj algorytm **Max** lub algorytm **Min**, by otrzymać algorytm rozwiązywania wybranego problemu z ćwicz. 1.



Rysunek 1.
Demonstracja działania algorytmu **Min** w programie **Maszyna sortująca**

Jeszcze jedna modyfikacja algorytmu Min

Często, poza znalezieniem elementu najmniejszego (lub największego) chcielibyśmy znać jego położnie, czyli miejsce (numer, indeks) w ciągu danych.

Ćwiczenie 5. Zmodyfikuj algorytm **Min** tak, aby wynikiem działania, oprócz wartości najmniejszego elementu, był również indeks, wskazujący na miejsce tego elementu w ciągu danych. Wprowadź w tym celu nową

zmienną, np. *imin,* w której będzie przechowywany numer aktualnie najmniejszego elementu.

Schematy blokowe algorytmu Min

Schemat blokowy algorytmu (zwany również **siecią działań** lub **siecią obliczeń**) jest graficznym opisem: działań składających się na algorytm, ich wzajemnych powiązań i kolejności ich wykonywania. W informatyce miejsce schematów blokowych jest pomiędzy opisem algorytmu w postaci listy kroków, a programem, napisanym w wybranym języku programowania. Należą one do kanonu wiedzy informatycznej, nie są jednak niezbędnym jej elementem, chociaż mogą okazać się bardzo przydatne na początkowym etapie projektowania algorytmów i programów komputerowych. Z drugiej strony, w wielu publikacjach algorytmy są przedstawiane w postaci schematów blokowych, pożądana jest więc umiejętność ich odczytywania i rozumienia. Warto nadmienić, że ten sposób reprezentowania algorytmów pojawia się w zadaniach maturalnych z informatyki.

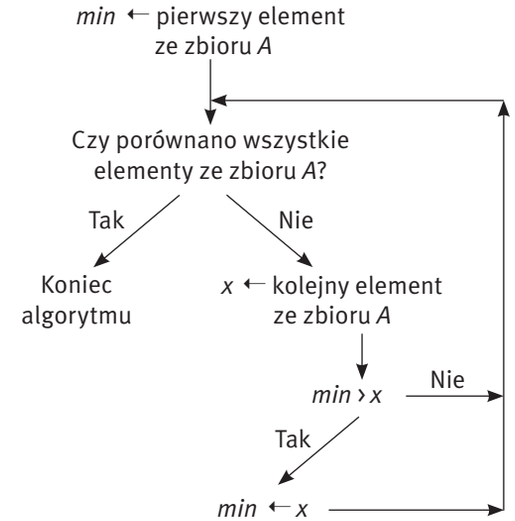
Na rys. 2 przedstawiono schemat algorytmu **Min.** Jest to bardziej schemat ideowy działania algorytmu, niż jego schemat blokowy, jest bardzo „zgrubny", gdyż zawarto w nim jedynie najważniejsze polecenia i pominięto szczegóły realizacji poszczególnych poleceń.

Na rys. 3 jest przedstawiony szczegółowy schemat blokowy algorytmu **Min**, uwzględniono w nim również modyfikację zaproponowaną w ćwicz. 5. Jest on zbudowany z bloków, których kształty zależą od rodzaju wykonywanych w nich poleceń. I tak mamy:

- blok początku i blok końca algorytmu;
- blok wprowadzania (wczytywania) danych i wyprowadzania (drukowania) wyników – bloki te mają taki sam kształt;
- blok operacyjny, w którym są wykonywane operacje przypisania;
- blok warunkowy, w którym jest formułowany warunek;
- blok informacyjny, który może służyć do komentowania fragmentów schematu lub łączenia ze sobą części większych schematów blokowych.

Nie istnieje pełny układ zasad poprawnego konstruowania schematów blokowych. Można natomiast wymienić dość naturalne zasady, wynikające z charakteru bloków:

- schemat zawiera dokładnie jeden blok początkowy, ale może zwierać wiele bloków końcowych – początek algorytmu jest jednoznacznie określony, ale algorytm może się kończyć na wiele różnych sposobów;
- z bloków: początkowego, wprowadzania danych, wyprowadzania wyników, operacyjnego wychodzi dokładnie jedno połączenie, może jednak wchodzić do nich wiele połączeń;
- z bloku warunkowego wychodzą dwa połączenia, oznaczone wartościami warunku: TAK i NIE;
- połączenia wychodzące mogą dochodzić do bloków lub do innych połączeń.



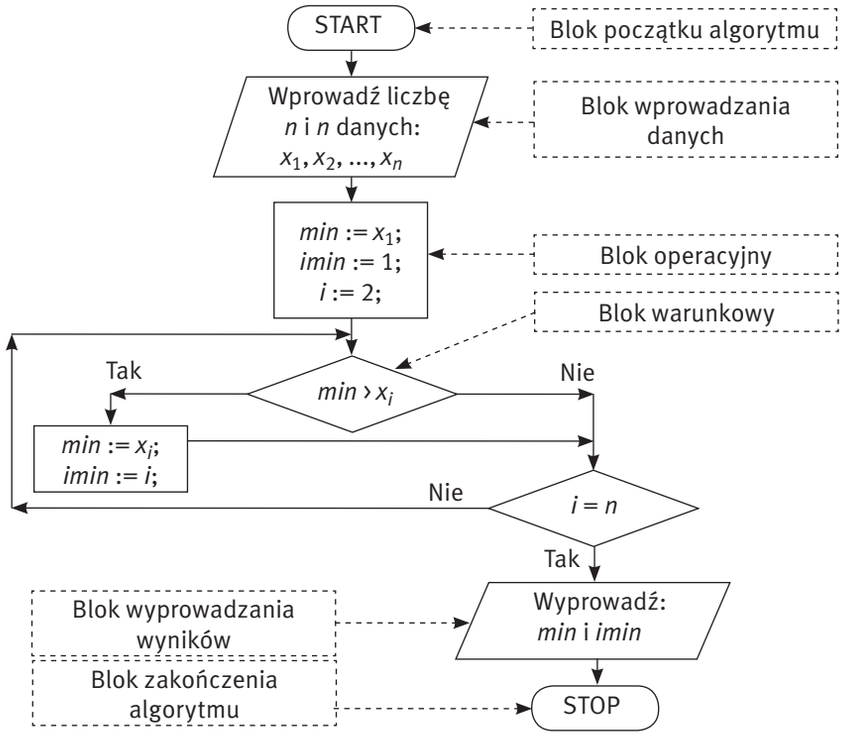
Rysunek 2.
Pierwszy, „zgrubny" schemat blokowy algorytmu **Min** – znajdowania najmniejszego elementu *min* w zbiorze elementów *A*

Ćwiczenie 6. Zakreśl na schemacie blokowym na rys. 3 fragmenty odpowiadające poszczególnym krokom w opisie algorytmu **Min**. Zauważ, ten że schemat blokowy zawiera również fragmenty odpowiadające danym i wynikom w specyfikacji algorytmu.

Ćwiczenie 7. Blok wprowadzania danych w schemacie na rys. 3 polega na wczytaniu liczby n a później na wczytaniu n liczb x_i. Narysuj szczegółowy schemat blokowy tego bloku.

Ćwiczenie 8. Zmodyfikuj schemat blokowy na rys. 3 tak, aby powstał schemat blokowy algorytmu **Max**.

Schematy blokowe mają wady, trudne do wyeliminowania. Łatwo konstruuje się z ich pomocą algorytmy dla obliczeń nie zawierających iteracji i warunków, którym w schematach odpowiadają rozgałęzienia, nieco trudniej dla obliczeń z rozgałęzieniami, a trudniej dla obliczeń iteracyjnych (wczytywanie ciągu i realizacja Kroku 2 z algorytmu **Min**). Za pomocą schematów blokowych nie można w naturalny sposób zapisać rekurencji oraz objaśnić znaczenia wielu pojęć związanych z algorytmiką, takich np. jak programowanie z użyciem procedur, czyli podprogramów z parametrami.



Rysunek 3.
Schemat blokowy algorytmu **Min**, zmodyfikowanego jak w ćwicz. 5. Bloków informacyjnych użyto do opisania innych bloków

W dalszej części zajęć rzadko będziemy korzystać ze schematów blokowych, wystarczy nam bowiem umiejętność programowania.

Skomputeryzowany schemat blokowy

Schematy blokowe doczekały się skomputeryzowanej wersji, na przykład w postaci programu **ELI**, który m.in. umożliwia wykonywanie na komputerze utworzonych w nim schematów. Dzięki temu jest możliwa komputerowa realizacja algorytmów, bez konieczności ich programowania. Jeśli wcześniej miałeś styczność z programem **ELI**, to wykonaj następne ćwiczenie.

Ćwiczenie 9. Utwórz sam, albo zapoznaj się z gotową realizacją schematu blokowego algorytmu **Min** lub **Max** wykonaną w programie **ELI**. Prześledź działanie tego schematu wybierając pracą krokową. Wykonaj ten schemat dla różnych danych.

Przykładowa realizacja ćwicz. 9 jest pokazana na rys. 4.

Słuchaczom zainteresowanym programem **ELI** polecamy książkę Algorytmy, w której większość omawianych algorytmów została zrealizowana w tym programie.

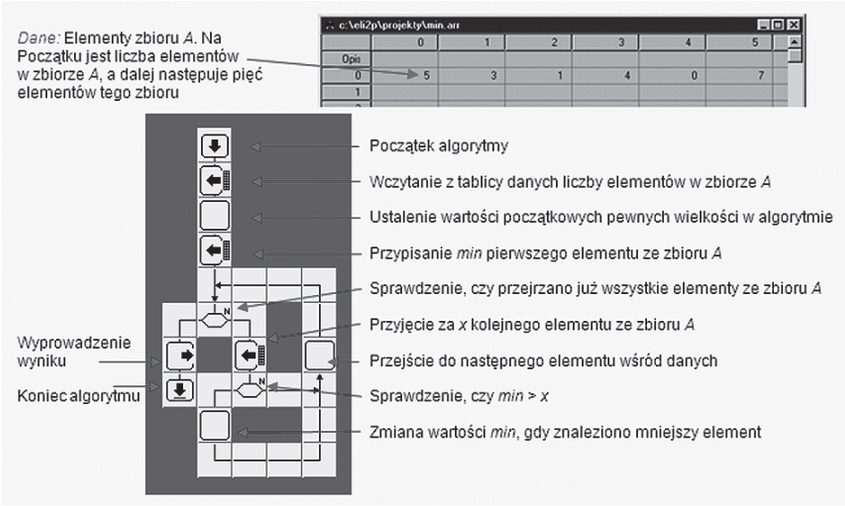
Reprezentowanie danych w algorytmach

Zanim podamy komputerową realizację pierwszego algorytmu, musimy ustalić, w jaki sposób będą reprezentowane w algorytmie dane i jak będziemy je podawać do algorytmu.

Wspomnieliśmy już przy projektowaniu algorytmu **Min**, że dane dla tego algorytmu są zapisane w postaci ciągu n liczb $x_1, x_2, \ldots, x_n$. Liczby te mogą być naturalne (czyli całkowite i dodatnie), całkowite lub dziesiętne (np. z kropką). Rodzaj danych liczb nazywa się **typem danych**. Przyjmujemy dla uproszczenia, że danymi dla algorytmów omawianych na tych zajęciach są liczby całkowite.

Zbiór danych, który jest przetwarzany za pomocą algorytmu, może być podawany (czytany) z klawiatury, czytany z pliku lub może być zapisany w innej strukturze danych.

Dla wygody będziemy zakładać, że wiemy, ile będzie danych i ta liczba danych występuje na początku danych – jest nią liczba n w opisie algorytmu **Min**. W ogólności, jeśli np. dane napływają do komputera z jakiegoś urządze-



Rysunek 4.
Schemat blokowy algorytmu **Min** zrealizowany w postaci projektu, utworzonego w programie **ELI**

nia pomiarowego, możemy nie wiedzieć, ile ich będzie. W takich przypadkach wprowadza się dane aż do specjalnego znaku, który świadczy o ich końcu. Takim znakiem może być koniec pliku, jeśli dane są umieszczone w pliku. Może nim być również wyróżniona liczba zwana **wartownikiem**, której rolą jest pilnowanie końca danych O użyciu wartownika powiemy później, a teraz przedstawimy dwie realizacje algorytmu **Min** dla dwóch sposobów podania zbioru danych przy założeniu, że na początku jest podawana liczba elementów w zbiorze danych.

Uwaga. Piszemy „zbiór danych", ale użycie tutaj pojęcia zbiór nie zawsze jest matematycznie poprawne. W zbiorze elementy się nie powtarzają, a w danych mogą występować takie same liczby. Formalnie, czyli całkowicie poprawnie powinniśmy mówić o tzw. **multizbiorach**, czyli zbiorach, w których elementy mogą się powtarzać, ale dla wygody będziemy stosować pojęcie zbioru, pamiętając, że mogą powtarzać się w nim elementy. Sytuację upraszcza nam założenie, że zbiór danych w algorytmie będzie przedstawiony w postaci ciągu elementów, a w ciągu elementy mogą się powtarzać.

Komputerowa realizacja algorytmu Min – dane z klawiatury

Zapiszemy teraz algorytm **Min** posługując się poleceniami języka Pascal. Odpowiednie realizacje przedstawionych algorytmów w języku C++ zostaną podane na zajęciach warsztatowych. Przyjmujemy na początku, że dane są podawane z klawiatury – na początku należy wpisać liczbę wszystkich danych, a po niej kolejne elementy danych w podanej ilości. Po każdej danej liczbie naciskamy klawisz Enter.

Program, który jest zapisem algorytmu **Min** w języku Pascal, jest umieszczony w drugiej kolumnie w tab. 1. Język Pascal jest zrozumiały dla komputerów, które są wyposażone w specjalny program, tzw. **kompilator**, przekładający programy użytkowników na język wewnętrzny komputerów. Program w tab. 1 bez większego trudu zrozumie także człowiek dysponujący opisem algorytmu **Min** w postaci listy kroków. Kilka tylko słów komentarza. W wierszu nr 2 znajdują się **deklaracje** zmiennych – komputer musi wiedzieć, jakimi wielkościami posługuje się algorytm i jakiego są one typu, `integer` oznacza liczby całkowite. Polecenia w językach programowania nazywają się **instrukcjami**. Jeśli chcemy z kilku instrukcji zrobić jedną, to tworzymy z nich **blok**, umieszczając na jego początku słowo `begin`, a na końcu – `end`. Pojedyncze instrukcje kończymy średnikiem. Na końcu programu stawiamy kropkę.

Dwie instrukcje wymagają wytłumaczenia, chociaż również są dość oczywiste. W wierszach 6 – 11 znajdują się instrukcje, które realizują Krok 2 algorytmu, polegający na wykonaniu wielokrotnie sprawdzenia warunku. Instrukcja, służąca do wielokrotnego wykonania innych instrukcji nazywa się **instrukcją iteracyjną** lub **instrukcją pętli**. W programie w tab. 1 ta instrukcja zaczyna się w wierszu nr 6 a kończy w wierszu nr 11:

Ada Augusta, córka Byrona, uznawana powszechnie za pierwszą programistkę komputerów, przełomowe znaczenie maszyny analitycznej Ch. Babbage'a, pierwowzoru dzisiejszych komputerów, upatrywała właśnie „w możliwości wielokrotnego wykonywania przez nią danego ciągu instrukcji, z liczbą powtórzeń z góry zadaną lub zależną od wyników obliczeń", a więc w iteracji.

```
for i:=2 to n do begin
...
end
```

Ta instrukcja iteracyjna służy do powtórzenia **instrukcji warunkowej**, która zaczyna się w wierszu nr 8 i kończy w wierszu nr 10. Ma tutaj postać:

```
if min>x then begin
...
end
```

Inne typy instrukcji iteracyjnej i warunkowej będą wprowadzane sukcesywnie.

Tabela 1.
Program w języku Pascal (druga kolumna)

Lp	Program w języku Pascal	Odpowiedniki instrukcji po polsku
1.	`Program MinKlawiatura;`	`nazwa programu;`
2.	`var i,imin,min,n,x:integer;`	`zmienne i,imin,min,n,x: naturalne;`
3.	`begin`	`początek`
4.	`read(n);`	`czytaj(n);`
5.	`read(x); min:=x; imin:=1;`	`czytaj(x); min:=x; imin:=1;`
6.	`for i:=2 to n do begin`	`dla i:=2 do n wykonaj begin`
7.	`read(x);`	`czytaj(x);`
8.	`if min>x then begin`	`jeśli min>x to początek`
9.	`min:=x; imin:=i`	`min:=x; imin:=i`
10.	`end`	`Koniec`
11.	`end;`	`koniec;`
12.	`write(imin,min)`	`drukuj(imin, min)`
13.	`end.`	`koniec. – na końcu stawiamy kropkę`

Zagłębiające się bloki instrukcji

Komputerowa realizacja algorytmu Min – dane z tablicy

W poniższej realizacji algorytmu **Min** przyjmujemy, że dane są najpierw umieszczane w tablicy, która ma *n* elementów, a następnie jest wykonywany algorytm[4].

```
Program MinTablica;
 var i,imin,min,n:integer;
     x:array[1..100] of integer;
     {Przyjmujemy, że dane mają co najwyżej 100 liczb.}
begin
 read(n);
 for i:=1 to n do read(x[i]);
 min:=x[1]; imin:=1;
 for i:=2 to n do
  if min>x[i] then begin min:=x[i]; imin:=i end;
 write(min,imin)
end.
```

Ćwiczenie 10. Uruchom oba programy służące do znajdowania najmniejszej liczby w ciągu i sprawdź ich poprawność w środowisku, które będzie wprowadzone na zajęciach warsztatowych. Dane podawaj z klawiatury.

Pracochłonność (złożoność) algorytmu Min

Problem znajdowania najmniejszego (lub największego) elementu w zbiorze jest jednym z elementarnych problemów najczęściej rozwiązywanych przez człowieka i przez komputer, dlatego interesujące jest pytanie, czy rozwiązujemy go możliwie najszybciej. W szczególności, czy podany przez nas algorytm **Min** i jego komputerowe **implementacje**[5] są najszybszymi metodami rozwiązywania tego problemu.

W algorytmach **Min** i **Max** i w ich implementacjach podstawową operacją jest porównanie dwóch elementów ze zbioru danych – policzmy więc, ile porównań jest wykonywanych w tych algorytmach. Liczba tych porównań w algorytmie zależy od liczby danych. W każdej iteracji algorytmu jest wykonywane jedno porównanie $min > x_i$, a zatem w każdym z tych algorytmów jest wykonywanych $n - 1$ porównań (tyle razy bowiem jest wykonywana iteracja w kroku 2). Pozostałe operacje służą głównie do organizacji obliczeń i ich liczba jest związana z liczbą porównań. Na przykład, operacja przypisania $min := x_i$ może być wykonana tylko o jeden raz więcej – w kroku 1 i $n - 1$ razy w kroku 2.

[4] W taki sam sposób są przechowywane dane w projekcie utworzonym w programie **ELI**, patrz rys. 4.

[5] Terminem **Implementacja** określa się w informatyce komputerową realizację algorytmu

Możemy więc podsumować nasze rozumowanie:

> najmniejszy (lub największy) element w niepustym zbiorze danych można znaleźć wykonując o jedno porównanie mniej niż wynosi liczba wszystkich elementów w tym zbiorze.

To nie jest specjalnie wielkie odkrycie, a jedynie sformułowanie dość oczywistej własności postępowania, które często wykonujemy niemal automatycznie, nie zastanawiając się nawet specjalnie, w jaki sposób to robimy. Już większym wyzwaniem jest pytanie:

Czy w zbiorze złożonym z n liczb, można znaleźć najmniejszy element wykonując mniej niż $n - 1$ porównań między elementami tego zbioru?

Udzielimy negatywnej odpowiedzi na to pytanie[6], posługując się interpretacją wziętą z klasowego turnieju tenisa (zob. ćwicz. 1). Ile należy rozegrać meczów (to są właśnie porównania w przypadku tego problemu), aby wyłonić najlepszego tenisistę w klasie? Lub inaczej – kiedy możemy powiedzieć, że Tomek jest w naszej klasie najlepszym tenisistą? Musimy mieć pewność, że wszyscy pozostali uczniowie są od niego gorsi, czyli przegrali z nim, bezpośrednio lub pośrednio. A zatem każdy inny uczeń przegrał przynajmniej jeden mecz, czyli musiano rozegrać przynajmniej tyle meczów, ilu jest uczniów w klasie mniej jeden. I to kończy nasze uzasadnienie.

Z dotychczasowych rozważań możemy wyciągnąć wniosek, że algorytmy **Min** i **Max** oraz ich komputerowe implementacje są najlepszymi algorytmami służącymi do znajdowania najmniejszego i największego elementu, gdyż wykonywanych jest w nich tyle porównań, ile musi wykonać jakikolwiek algorytm rozwiązywania tych problemów. O takim algorytmie mówimy, że jest **algorytmem optymalnym pod względem złożoności obliczeniowej**.

Pamiętaj. Poszukiwanie elementu w zbiorze nieuporządkowanym za pomocą metody przeglądania kolejnych elementów, jest postępowaniem optymalnym, czyli najszybszym wśród możliwych.

[6] Posługujemy się tutaj argumentacją zaczerpniętą z książki Hugona Steinhausa, *Kalejdoskop matematyczny* (WSiP, Warszawa 1989, rozdz. III), [8].

Naszkicujemy bardziej sformalizowany dowód optymalności algorytmu **Min**, podany przez Ira Pohl w 1972 roku. Oznaczmy przez X zbiór n elementów, wśród których chcemy znaleźć element najmniejszy – oznaczany dalej przez *min*. Ograniczamy uwagę do algorytmów, w których jako podstawowe działanie jest wykonywane porównanie między elementami zbioru X.

Oznaczmy przez (A, B) parę zbiorów o tej własności, że zbiór A zawiera *min* i $B = X - A$. A zatem, przed rozpoczęciem działania algorytmu znajdowania *min* mamy $(A, B) = (X,\emptyset)$ a po zakończeniu – chcemy, aby $(A, B) = (\{min\}, X - \{min\})$. Tej transformacji pary zbiorów odpowiada zmiana ich liczebności: $(n, 0) \rightarrow (1, n - 1)$. Możemy więc powiedzieć, że jakikolwiek algorytm znajdujący *min*, przeprowadza parę zbiorów (A, B) o liczebnościach $(n, 0)$ w parę o liczebnościach $(1, n - 1)$, stosując jedynie porównania między elementami tych zbiorów. Określmy więc, jaki wpływ na parę zbiorów (A, B) i ich liczebności (k, l), gdzie $k + l = n$, ma wykonanie jednego porównania dwóch elementów należących do tych zbiorów. Możliwe są cztery typy porównań, w których $a, a' \in A$ i $b, b' \in B$:

$a < a'$ $\quad (k, l) \rightarrow (k - 1, l + 1)$,
$a < b$ $\quad (k, l) \rightarrow (k, l)$ lub $(k - 1, l + 1)$,
$b < a$ $\quad (k, l) \rightarrow (k, l)$ lub $(k - 1, l + 1)$,
$b < b'$ $\quad (k, l) \rightarrow (k, l)$.

W pierwszym przypadku, bez względu na prawdziwość nierówności, jeden z elementów można przenieść ze zbioru A do zbioru B. A w drugim i trzecim przypadku – niekoniecznie. W ostatnim zaś przypadku, porównanie między elemenatmi zbioru B nie ma wpływu na stan zbioru A.

Stąd wynika, że największe zmiany liczebności pary zbiorów w jednym kroku zachodzą podczas wykonywania pierwszego typu porównań i takich kroków trzeba wykonać $n - 1$, by przejść od stanu $(n, 0)$ do stanu $(1, n - 1)$.

Uzasadniliśmy więc, że znalezienie elementu *min* w zbiorze złożonym z n elementów wymaga wykonania przynajmniej $n - 1$ porównań między elementami tego zbioru, czyli algorytm *Min* jest optymalny, gdyż wykonuje dokładnie taką liczbę porównań.

3 KOMPLETOWANIE PODIUM ZWYCIĘZCÓW TURNIEJU

Przedstawione w poprzednim rozdziale postępowanie nie jest jedyną metodą służącą do znajdowania najlepszego elementu w zbiorze. Inną metodą jest tzw. **system pucharowy**, stosowany często przy wyłanianiu najlepszego zawodnika bądź drużyny w turnieju. W metodzie tej „porównanie" dwóch zawodników (lub drużyn), by stwier-

dzić, który jest lepszy („większy”), polega na rozegraniu meczu. W rozgrywkach systemem pucharowym zakłada się, że wszystkie mecze kończą się zwycięstwem jednego z zawodników, dlatego w dalszej części będziemy pisać o rozgrywkach w tenisa, a nie o turnieju w warcaby, gdyż w przypadku warcabów (jak i szachów) partie mogą kończyć się remisem, podczas gdy w meczach w tenisa można wymóc, by mecz między dwoma zawodnikami zawsze kończył się zwycięstwem jednego z nich.

Wyłanianie zwycięzcy w turnieju

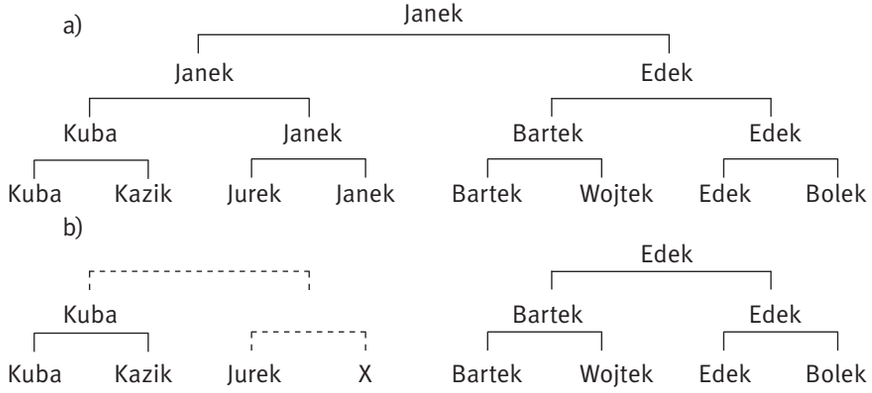
Nurtować może pytanie, czy znajdowanie najlepszego zawodnika systemem pucharowym nie jest czasem metodą bardziej efektywna pod względem liczby wykonywanych porównań (czyli rozegranych meczy), niż przeszukiwanie liniowe, opisane w poprzednim rozdziale.

Na rys. 4(a) jest przedstawiony fragment turnieju, rozegranego między ośmioma zawodnikami. Zwycięzcą okazał się Janek po rozegraniu w całym turnieju siedmiu meczów. A zatem podobnie jak w przypadku metody liniowej, aby wyłonić zwycięzcę, czyli najlepszego zawodnika (elementu) wśród ośmiu zawodników, należało rozegrać o jeden mecz mnie niż wystąpiło w turnieju zawodników. Nie jest to przypadek. Ten fakt jest prawdziwy dla dowolnej liczby zawodników występujących w turnieju, rozgrywanym metodą pucharową.

Ćwiczenie 11. Narysuj drzewa rozgrywek pucharowych dla różnej liczby zawodników, np.: 6. 7. 9. 10. 11. 12. 13. 14. 15 i przekonaj się, ze w każdym przypadku liczba rozegranych meczów jest o jeden mniejsza od liczby zawodników biorących udział w turnieju.

Powyższa prawidłowość wynika z następującego faktu: schemat turnieju jest drzewem binarnym, a takim drzewie liczba wierzchołków pośrednich jest o jeden mniejsza od liczby wierzchołków końcowych. Wierzchołki końcowe to zawodnicy przystępujący do turnieju, a wierzchołki pośrednie odpowiadają rozegranym meczom. Następne ćwiczenie może być nieco trudniejsze dla początkującego programisty, pozostawiamy je więc dla ambitniejszych uczniów.

Ćwiczenie 12. Zapisz w postaci listy kroków algorytm służący do znajdowania największej liczby w ciągu „metodą pucharową”; dany ciąg może mieć dowolną długość, nie koniecznie będąca potęgą liczby 2. Przyjmij, że na początku ciąg liczb jest dany w tablicy i postaraj się, by w trakcie algorytmu pozostałe z ciągu liczby także były pamiętane w tej samej tablicy. Zaimplementuj opisany algorytm w wybranym języku programowania.



Rysunek 5.
Drzewo przykładowych rozgrywek w turnieju tenisowym (a)
oraz drzewo znajdowania drugiego najlepszego zawodnika turnieju (b)

Wyłanianie drugiego najlepszego zawodnika turnieju

Bardzo ciekawy problem postawił około 1930 roku Hugo Steinhaus. Zastanawiał się on bowiem, jaka jest najmniejsza liczba meczów tenisowych do rozegrania w grupie zawodników, niezbędna do tego, aby wyłonić wśród nich najlepszego i drugiego najlepszego zawodnika. Wtedy, tak jak i dzisiaj, rozgrywano turnieje tenisowe na ogół systemem pucharowym. Zapewnia on, że zwycięzca finału jest najlepszym zawodnikiem, gdyż pokonał wszystkich uczestników turnieju: niektórych bezpośrednio – wygrywając z nimi w spotkaniach, a niektórych pośrednio – pokonując ich zwycięzców. W takich turniejach drugą nagrodę otrzymuje zwykle zawodnik pokonany w finale. I tutaj Steinhaus miał słuszne wątpliwości, czy jest to właściwa decyzja, tzn., czy pokonany w finale jest drugim najlepszym zawodnikiem turnieju, czyli czy jest lepszy od wszystkich pozostałych zawodników z wyjątkiem zwycięzcy turnieju.

By się przekonać, że wątpliwości H. Steinhausa były rzeczywiście uzasadnione, spójrzmy na drzewo turnieju przedstawione na rys. 5(a). Zwycięzcą w tym turnieju jest Janek, który w finale pokonał Edka. Edkowi przyznano więc drugą nagrodę, chociaż wykazał, że jest lepszy jedynie od Bolka, Bartka i Wojtka (gdyż przegrał z Bartkiem). Nic nie wiemy, jakby Edek grał przeciwko zawodnikom z poddrzewa, z którego jako zwycięzca został wyłoniony Janek. Jak można naprawić ten błąd organizatorów rozgrywek tenisowych? Istnieje prosty sposób znalezienia drugiego najlepszego zawodnika turnieju – rozegrać jeszcze jedną pełną rundę z pominięciem zwycięzcy turnieju głównego. Wówczas, najlepszy i drugi najlepszy zawodnik zawodów zostaliby wyłonieni w $(n-1)+(n-2)=2n-3$ meczach. Hugo Steinhaus oczywiście znał to rozwiązanie, pytał więc o najmniejszą potrzebną liczbę meczów, i takiej odpowiedzi udzielił w 1932 inny polski matematyk J. Schreier, chociaż jego dowód nie był w pełni poprawny i został skorygowany dopiero po 32 latach (w 1964 roku przez S.S. Kislicyna).

Pamiętaj. Wicemistrz wyłoniony systemem pucharowym na ogół nie jest drugą najlepszą drużyną (zawodnikiem) turnieju.

Jeśli chcemy, aby drugi najlepszy zawodnik nie musiał być wyłaniany w nowym pełnym turnieju, to musimy umieć skorzystać ze wszystkich wyników głównego turnieju. Posłużymy się drzewem turnieju z rys. 5(a). Zauważmy, że Edek jest oczywiście najlepszy wśród zawodników, którzy w drzewie rozgrywek znajdują się w wierzchołkach leżących poniżej najwyższego wierzchołka, który on zajmuje. Musimy więc jedynie porównać go z zawodnikami drugiego poddrzewa. Aby i w tym poddrzewie wykorzystać wyniki dotychczasowych meczów, eliminujemy z niego Janka — zwycięzcę turnieju i wstawiamy Edka na jego początkowe miejsce X. Spowoduje to, że Edek zostanie porównany z najlepszymi zawodnikami w drugim poddrzewie. Na rys. 5(b) oznaczyliśmy przerywaną linią mecze, które zostaną rozegrane w tej części turnieju – Jurek z Edkiem i zwycięzca tego meczu z Kubą, a więc dwa dodatkowe mecze.

Algorytm ten można, po zmianie słownictwa, zastosować do znajdowania największej i drugiej największej liczby w zbiorze danych.

Złożoność wyłaniania zwycięzcy i drugiego najlepszego zawodnika turnieju

Ile porównań jest wykonywanych w opisanym algorytmie znajdowania najlepszego i drugiego najlepszego zawodnika w turnieju? Najlepszy zawodnik jest wyłaniany w $n-1$ meczach, gdzie n jest liczbą wszystkich zawodników. Z kolei, aby wyłonić drugiego najlepszego zawodnika, trzeba rozegrać tyle meczów, ile jest poziomów w drzewie turnieju głównego (z wyjątkiem pierwszego poziomu). A zatem, jaka jest wysokość drzewa turnieju? Dla uproszczenia przyjmijmy, że drzewo jest **pełne**, tzn. każdy zawodnik ma parę, czyli w każdej rundzie turnieju gra parzysta liczba zawodników. Stąd wynika, że na najwyższym poziomie jest jeden zawodnik, na poziomie niższym – dwóch, na kolejnym – czterech itd. Czyli, liczba zawodników rozpoczynających turniej jest potęgą liczby 2, zatem $n=2^k$, gdzie k jest liczbą poziomów drzewa — oznaczmy ją przez $\log_2 n$. Algorytm wykonuje więc $(n-1)+(\log_2 n-1)=n+\log_2 n-2$ porównań. Jeśli n nie jest potęgą liczby 2, to na ogół w turnieju niektórzy zawodnicy otrzymują wolną kartę, a podana liczba jest oszacowaniem z góry liczby rozegranych meczów.

Ćwiczenie 13. Porównaj wartości dwóch wyrażeń: $2n-3$ oraz $n+\log_2 n-2$, odpowiadających liczbie porównań wykonywanych w dwóch, omówionych wyżej algorytmach znajdowania najlepszego i drugiego najlepszego zawodnika turnieju. Dla ułatwienia, obliczenia wykonaj dla n będących potęgami liczby 2.

Dodajmy tutaj, że przedstawiony powyżej algorytm znajdowania najlepszego i drugiego najlepszego zawodnika turnieju jest optymalny, tzn. najszybszy w sensie liczby rozegranych meczów (porównań).

Na naszym podium zwycięzców turnieju tenisowego brakuje jeszcze trzeciego najlepszego zawodnika, czyli kogoś, kto jest lepszy od wszystkich pozostałych zawodników z wyjątkiem już wyłonionych — najlepszego i drugiego najlepszego. Znalezienie go bardzo przypomina wyłanianie drugiego najlepszego zawodnika.

Ćwiczenie 14. Przypuśćmy, że w naszym przykładowym turnieju, drugie miejsce zajął Kuba. W jaki sposób należy zorganizować dogrywkę, by wyłonić trzeciego najlepszego zawodnika turnieju. A jeśli drugie miejsce zajął jednak Edek — jak należy postępować w tym przypadku? Sformułuj ogólną zasadę. Ile meczów należy rozegrać, by wyłonić zawodnika zajmującego trzecie miejsce?

To postępowanie można kontynuować wyznaczając czwartego, piątego itd. zawodnika turnieju. Ostatecznie otrzymamy pełne uporządkowanie wszystkich zawodników biorących udział w turnieju. Taka metoda nazywa się **porządkowaniem na drzewie** (patrz [6]) i może być stosowana również do porządkowania liczb.

4 JEDNOCZESNE ZNAJDOWANIE NAJMNIEJSZEGO I NAJWIĘKSZEGO ELEMENTU

Jedną z miar, określającą, jak bardzo są porozrzucane wartości obserwowanej w doświadczeniu wielkości, jest **rozpiętość** zbioru, czyli różnica między największą (w skrócie, maksimum) a najmniejszą wartością elementu (w skrócie, minimum) w zbiorze. Im większa jest rozpiętość, tym większy jest rozrzut wartości elementów zbioru. Interesujące jest więc jednoczesne znalezienie najmniejszej i największej wartości w zbiorze liczb.

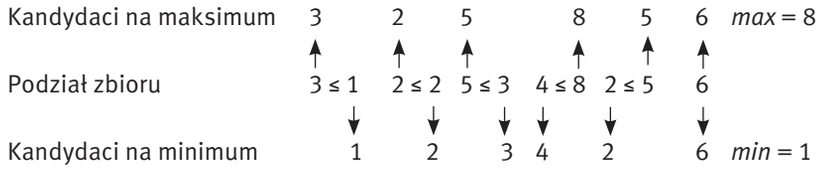
Rozwiązanie naiwne

Ćwiczenie 15. Na podstawie dotychczasowych rozważań, dotyczących wyznaczania najmniejszej i największej wartości w zbiorze liczb, zapewne łatwo podasz algorytm znajdowania jednocześnie obu tych elementów w zbiorze. Ile należy w tym celu wykonać porównań?

Rozwiązanie tego ćwiczenia ilustruje częste podejście, stosowane w matematyce i informatyce, które polega na tym, że w rozwiązaniu nowego problemu korzystamy ze znanej już metody. Stosujemy więc najpierw algorytm **Min** do całego zbioru, a później algorytm **Max** do zbioru z usuniętym minimum. W takim algorytmie „jednoczesnego wyznaczania" minimum i maksimum w ciągu złożonym z n liczb jest wykonywanych $(n - 1) + (n - 2) = 2n - 3$ porównań.

Rozwiązanie bardziej przemyślane

Postaramy się znacznie przyspieszyć to postępowanie, a będzie to polegało na rzeczywiście jednoczesnym szukaniu najmniejszego i największego elementu w całym zbiorze, jak również wykorzystaniu poznanego algorytmu znajdowaniu tych elementów w pewnych podzbiorach rozważanego zbioru. W tym celu rozważmy ponownie podstawową operację – porównanie elementów – i zauważmy, że jeśli dwie liczby x i y spełniają nierówność $x \le y$, to x jest kandydatem na najmniejszą liczbę w zbiorze, a y jest kandydatem na największą liczbę w zbiorze. (Jeśli prawdziwa jest nierówność odwrotna, to wnioskujemy odwrotnie.) A zatem, porównując elementy parami, można podzielić dany zbiór elementów na dwa podzbiory, kandydatów na minimum i kandydatów na maksimum, i w tych zbiorach – które są niemal o połowę mniejsze niż oryginalny zbiór! – szukać odpowiednio minimum i maksimum. Pewnym problemem jest to, co zrobić z ostatnim elementem ciągu, gdy zbiór ma nieparzystą liczbę elementów. W tym przypadku decydujemy się dodać ten element do jednego i do drugiego podzbioru kandydatów. Postępowanie to jest zilustrowane przykładem na rys. 6.



Rysunek 6.
Przykład postępowania podczas jednoczesnego znajdowania minimum i maksimum w ciągu liczb

Zapiszmy opisane postępowanie w postaci algorytmu poprzedzając go specyfikacją.

Algorytm Min-i-Max – jednoczesne znajdowanie największego i najmniejszego elementu w zbiorze

Dane: Liczba naturalna n i zbiór n liczb dany w postaci ciągu $x_1, x_2, \ldots, x_n$.
Wynik: Najmniejsza liczba *min* i największa liczba *max* wśród liczb $x_1, x_2, \ldots, x_n$.
Krok 1. {Podział zbioru danych na dwa podzbiory: M – zbiór kandydatów na minimum i N – zbiór kandydatów na maksimum.
Na początku te zbiory są puste.}
Jeśli n jest liczbą parzystą, to dla $i = 1, 3, \ldots, n - 1$, a jeśli n jest liczbą nieparzystą, to dla $i = 1, 3, \ldots, n - 2$ wykonaj:
jeśli $x_i \le x_{i+1}$, to dołącz x_i do M, a x_{i+1} do N,
a w przeciwnym razie dołącz x_i do N, a x_{i+1} do M.
Jeśli n jest liczbą nieparzystą, to dołącz x_n do obu zbiorów M i N.
Krok 2. Znajdź *min* w zbiorze M, stosując algorytm **Min**.
Krok 3. Znajdź *max* w zbiorze N, stosując algorytm **Max**.

Metoda dziel i zwyciężaj

Ten algorytm jest przykładem metody, leżącej u podstaw wielu bardzo efektywnych algorytmów. Można w nim wyróżnić dwa etapy:

- podziału danych na dwa podzbiory równoliczne (Krok 1);
- zastosowanie znanych już algorytmów **Min** i **Max** do utworzonych podzbiorów danych (Kroki 2 i 3).

Jest to przykład zasady (metody) rozwiązywania problemów. która jeszcze wielokrotnie pojawi się w tym podręczniku. Nosi ona nazwę **dziel i zwyciężaj** i jest jedną z najefektywniejszych metod algorytmicznych w informatyce. **Dziel** – odnosi się do podziału zbioru danych na podzbiory, zwykle o jednakowej liczbie elementów, do których następnie są stosowane odpowiednie algorytmy. **Zwycięstwo** – to efekt końcowy, czyli efektywne rozwiązanie rozważanego problemu. Można się o tym przekonać wykonując następne ćwiczenie.

Nazwa zasady **dziel i zwyciężaj** pochodzi od angielskich słów *divide* and *conquer*. Nie należy jej jednak mylić z podobnie brzmiącą starożytną zasadą dziel i rządź (łac. *divide et impera*), która odnosiła się do sposobu rządzenia Cesarstwem Rzymskim, polegającego na dzieleniu wielkich obszarów i społeczeństw na mniejsze części, które w ten sposób miały utrudnioną komunikację między sobą, stanowiły więc mniejsze zagrożenie dla cesarzy. W przypadku zaś zasady **dziel i zwyciężaj** celem jest taki podział problemu na mniejsze części, by ich rozwiązania złożyły się na jak najefektywniejsze rozwiązanie głównego problemu.

Pracochłonność jednoczesnego znajdowania minimum i maksimum

Ćwiczenie 16. Oblicz, ile porównań między elementami danych jest wykonywanych w algorytmie **Min-i-Max**. Rozważ najpierw przypadek, gdy n jest liczbą parzystą, a następnie – gdy n jest liczbą nieparzystą.

Sprawdź teraz, że Twoje rozwiązanie ćwicz. 15 ma ogólną postać $\lceil 3n/2 \rceil - 2$, gdzie $\lceil x \rceil$oznacza tzw. powałę liczby[7], czyli najmniejszą liczbę całkowitą k spełniającą nierówność $x \leq k$. A zatem jest wykonywanych około $3n/2 - 2$ porównania, czyli ok. $n/2$ mniej porównań niż w algorytmie zasygnalizowanym w ćwicz. 15.

[7] Funkcja **powała** (i towarzysząca jej funkcja **podłoga**) odgrywają ważną rolę w rozważaniach informatycznych.

Implementacja algorytmu Min-i-Max

Ćwiczenie 17. Napisz program będący implementacją algorytmu **Min-i-Max**. Postaraj się, by Twój program wykonywał dokładnie $\lceil 3n/2 \rceil - 2$ porównania. Przekonaj się, że tak jest rozważając osobno n parzyste i n nieparzyste.

Wskazówka. W opisie algorytmu występują dwa podzbiory M i N, w których są umieszczane odpowiednio elementy będące kandydatami na minimum i na maksimum. Te zbiory możesz tworzyć w miejscu tablicy x, na przykład elementy zbioru M mogą być pamiętane na miejscach o parzystych indeksach, a elementy zbioru N na miejscach o nieparzystych indeksach. Wtedy w krokach 2 i 3 musisz szukać elementu najmniejszego i elementu największego na co drugich pozycjach w tablicy x.

Rekurencyjna realizacja metody dziel i zwyciężaj

Zastosowana tutaj zasada dziel i zwyciężaj jest na ogół stosowana w sposób **rekurencyjny** – problem jest dzielony na podproblemy, te są ponownie dzielone na podproblemy, i tak dalej, aż do otrzymania podproblemów, dla których rozwiązanie można łatwo wskazać, np. gdy liczba danych w podproblemie wynosi jeden lub dwa. Uczniom, którzy znają rekurencję, proponujemy rozwiązanie następnego ćwiczenia.

Ćwiczenie 18. Podaj opis rekurencyjnego algorytmu **Min-i-Max_rec**, który służy do jednoczesnego znajdowania najmniejszej i największej liczby w zbiorze liczb. Następnie wykonaj implementację tego algorytmu w języku programowania. Jeśli znasz sposób formułowania zależności rekurencyjnych, wyprowadź zależność na liczbę porównań w tym algorytmie i rozwiąż ją. Nie powinno Cię zdziwić, że w algorytmie rekurencyjnym jest wykonywanych również dokładnie $\lceil 3n/2 \rceil - 2$ porównań.

Nieco trudniej, niż w przypadku algorytmu znajdowania minimum, można również wykazać, że algorytmy **Min-i-Max** i **Min-i-Max_rec** są optymalne, tzn. wykonują możliwie najmniejszą liczbę porównań.

Ciekawe zadania

Na I i II Olimpiadzie Informatycznej (patrz materiały z tych olimpiad [1] i [2]) pojawiły się dwa zadania związane z istnieniem trójkątów o zadanych długościach boków. Jak wiemy, z trzech odcinków można zbudować trójkąt wtedy i tylko wtedy, gdy suma długości każdych dwóch odcinków jest większa od długości trzeciego odcinka – ten warunek nazywa się **warunkiem trójkatą**. W swych pełnych, konkursowych sformułowaniach, zadania te są dość trudne. Przedstawiamy je tutaj w nieco zmodyfikowanej postaci z uwagami, jak je rozwiązać. Szczegółowe omówienie tych zadań i ich rozwiązań znajduje się w cytowanych materiałach.

Zadanie A. W pliku dany jest skończony, co najmniej trzyelementowy zbiór A odcinków o długościach będących dodatnimi liczbami całkowitymi. Napisz program, który drukuje odpowiedź TAK, jeśli z każdych trzech odcinków ze zbioru *A* można zbudować trójkąt, lub odpowiedź NIE – w przeciwnym przypadku.

Wskazówka. W rozwiązaniu tego zadania należy najpierw przeformułować warunek trójkąta dla przypadku uporządkowanych długości jego boków, a następnie jednocześnie szukać najmniejszego i największego element w ciągu liczb, wczytywanym z pliku. To ostatnie założenie jest związane z tym, że plik danych, ze względu na swoja wielkość, może nie mieścić się w pamięci komputera.

Zadanie B. W pliku dany jest ciąg przynajmniej trzech liczb całkowitych dodatnich, nie większych niż miliard (tj. nie większych niż $10^9 = 1000000000$). Ułóż program, który sprawdza, czy wśród odcinków o długościach zapisanych w tym pliku istnieją trzy takie, z których można zbudować trójkąt.

Wskazówka. To zadanie tylko pozornie jest podobne do poprzedniego – jedyne ich podobieństwo ogranicza się do wykorzystania warunku trójkąta w rozwiązaniu. Rozwiązanie tego zadania może być pewnym zaskoczeniem dla uczniów – pojawiają się w nim bowiem **liczby Fibonacciego**.

5. PORZĄDKOWANIE PRZEZ WYBÓR – ITERACJA ALGORYTMU

Porządkowanie, nazywane również często **sortowaniem**, ma olbrzymie znaczenie niemal w każdej działalności człowieka. Jeśli elementy w zbiorze są uporządkowane zgodnie z jakąś regułą (np. książki lub ich karty katalogowe według liter alfabetu, słowa w encyklopedii, daty, numery telefonów według nazwisk właścicieli), to wykonywanie wielu operacji na tym zbiorze staje się znacznie łatwiejsze. Między innymi dotyczy to operacji:

- sprawdzenia czy dany element, czyli element o ustalonej wartości cechy, według której zbiór został uporządkowany, znajduje się w zbiorze,
- znalezienia elementu w zbiorze, jeśli w nim jest,
- dołączenia nowego elementu w odpowiednie miejsce, aby zbiór pozostał nadal uporządkowany.

Komputery w dużym stopniu zawdzięczają swoją szybkość temu, że działają na uporządkowanych informacjach. To samo odnosi się do nas – ludzi, gdy posługujemy się nimi, informacjami i komputerami. Jeśli chcemy na przykład sprawdzić, czy w jakimś katalogu dyskowym znajduje się plik o podanej nazwie, rozszerzeniu, czasie utworzenia lub rozmiarze, to najpierw odpowiednio porządkujemy listę plików (np. w programie Eksplorator Windows) i wtedy na ogół znajdujemy odpowiedź natychmiast. Porządkowanie jest również podstawową operacją wykonywaną na dużych zbiorach informacji, np. w bazach danych.

Często porządkujemy różne elementy lub wykonujemy powyższe operacje na uporządkowanych zbiorach nie korzystając z komputera – w tym również mogą nam pomóc metody porządkowania i algorytmy działające na uporządkowanych zbiorach omówione na zajęciach komputerowych.

Specyfikacja problemu porządkowania

Na tych zajęciach będziemy zajmować się głównie porządkowaniem liczb, chociaż wiele praktycznych problemów dotyczy porządkowania innych obiektów przechowywanych w komputerze. Przyjmijmy więc następującą specyfikację tego problemu.

Problem porządkowania (sortowania)

Dane: Liczba naturalna n i ciąg n liczb $x_1, x_2, \ldots, x_n$

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Z założenia, że porządkujemy tylko liczby względem ich wartości wynika, że interesują nas algorytmy, w których główną operacją jest porównanie, wykonywane między elementami danych.

Porządkowanie kilku elementów

Jeśli liczba elementów w ciągu jest mała, np. $n = 2, 3, 4$, to łatwo można podać algorytmy, w których jest wykonywana możliwie najmniejsza liczba porównań. Nieco bardziej złożonym problemem jest porządkowanie pięciu liczb w sposób optymalny. Pozostawiamy te szczególne przypadki do samodzielnego wykonania.

Ćwiczenie 19. Podaj w postaci drzewa porównań algorytm porządkowania 3 dowolnych liczb, a następnie rozszerz go na algorytm porządkowania 4 dowolnych liczb. Czy Twoje algorytmy wykonują odpowiednio nie więcej niż 3 i 5 porównań w najgorszym przypadku danych? Jeśli tak, to są to możliwie najlepsze algorytmy porządkowania 3 i 5 liczb. W podobny sposób nie da się otrzymać algorytmu porządkowania 5 liczb – zainteresowanych słuchaczy odsyłamy do książki [9], p. 4.3.

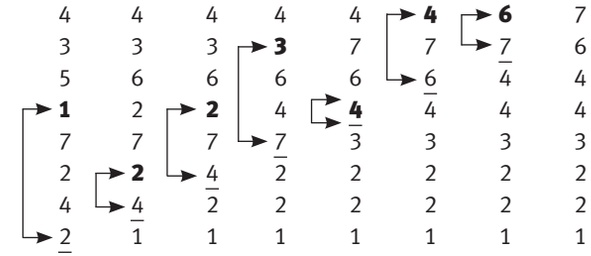
Porządkowanie dowolnej liczby elementów

Zajmiemy się teraz porządkowaniem ciągów, które mogą zawierać dowolną liczbę elementów. Wykorzystamy w tym celu jeden z poznanych wcześniej algorytmów. O innych algorytmach porządkowania wspominamy na końcu tego rozdziału – można zapoznać się z ich działaniem posługując się programem demonstracyjnym. Niektóre z nich przedstawiamy szerzej przy okazji omawiania wybranych technik algorytmicznych, w szczególności dotyczących rekurencji.

Jeden z najprostszych algorytmów porządkowania można wyprowadzić korzystając z tego, co już poznaliśmy w poprzednich punktach. Zauważmy, że: jeśli mamy ustawić elementy w kolejności od najmniejszego do największego, to najmniejszy element w zbiorze powinien się znaleźć na początku tworzonego ciągu, za nim powinien być umieszczony najmniejszy element w zbiorze pozostałym po usunięciu najmniejszego elementu itd. Taki algorytm jest więc **iteracją** znanego algorytmu znajdowania **Min** w ciągu i nosi nazwę **algorytmu porządkowania przez wybór.**

Demonstracja działania porządkowania przez wybór

Aby zilustrować działanie tego algorytmu załóżmy, że ciąg elementów, który mamy uporządkować, jest zapisany w kolumnie (zob. rys. 7). Chcemy ponadto, aby wynik, czyli ciąg uporządkowany, znalazł się w tym samym ciągu – o takim algorytmie mówimy, że działa *in situ*, czyli „w miejscu". W tym celu wystarczy znaleziony najmniejszy element w ciągu zamienić miejscami z pierwszym elementem tego ciągu – zob. rys. 7 ilustrujący kolejne kroki działania algorytmu porządkowania przez wybór.

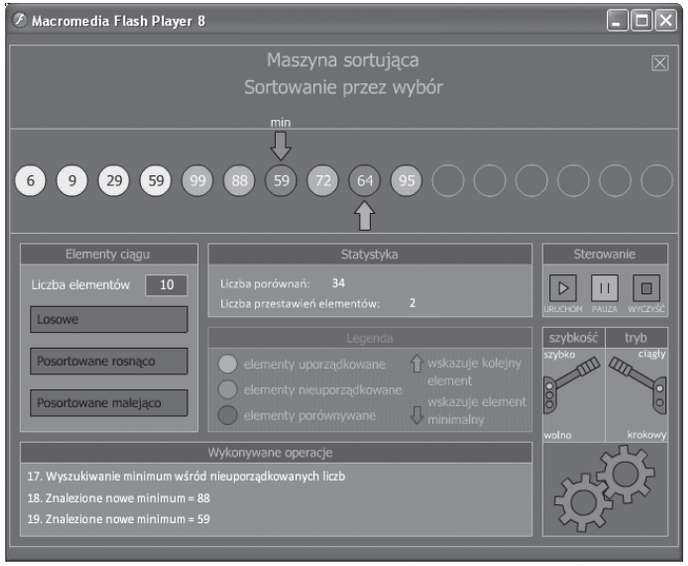


Rysunek 7.
Ilustracja działania algorytmu porządkowania przez wybór. W każdej kolumnie, pogrubiony został najmniejszy element w podciągu od góry do kreski, a klamra wskazuje zamianę elementów miejscami

Zanim podamy szczegółowy opis tego algorytmu porządkowania, przyjrzyj się pełnej demonstracji jego działania w programie **Maszyna sortująca**, który był wykorzystany do demonstracji działania algorytmu znajdującego najmniejszy element w ciągu.

Ćwiczenie 20. Posłuż się programem **Maszyna sortująca** (patrz rys. 8), by obejrzeć, jak działa algorytm porządkowania przez wybór. Zastosuj go do ciągów różnej długości i złożonych z elementów: losowych, uporządkowanych i odwrotnie uporządkowanych.

Polecamy również inny program **Sortowanie**, który służący do demonstracji działania oraz porównywania między sobą wielu algorytmów porządkujących. Ten program jest również załączony do materiałów tych zajęć i może być wykorzystany w celach edukacyjnych.

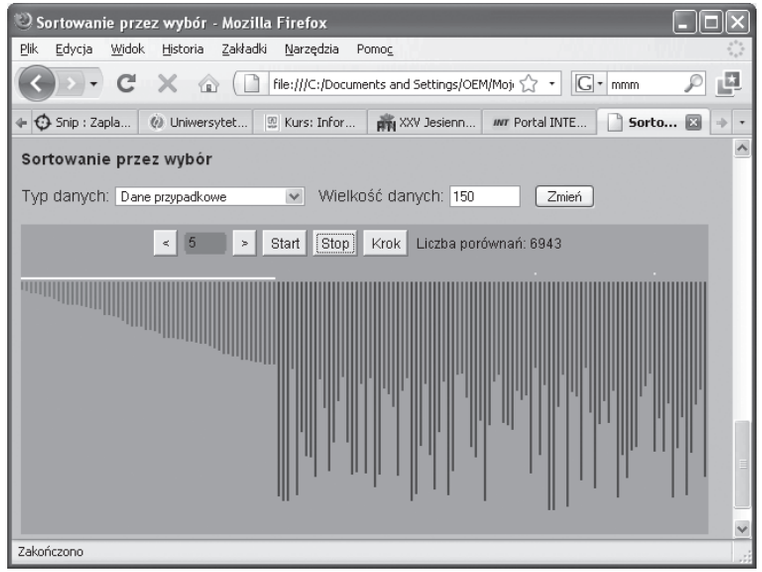


Rysunek 8.
Demonstracja działania algorytmu porządkowania przez wybór w programie **Maszyna sortująca** – cztery pierwsze elementy znajdują się już na swoim miejscu w ciągu uporządkowanym i szukany jest najmniejszy element w pozostałej części ciągu, by przenieść go na miejsce piąte

Ćwiczenie 21. Posłuż się programem **Sortowanie** (patrz rys. 9), by obejrzeć, jak działa algorytm porządkowania przez wybór. Zastosuj go do ciągów różnej długości i złożonych z elementów: losowych, uporządkowanych i odwrotnie uporządkowanych. Wykonaj demonstrację w pracy krokowej i ciągłej. Zauważ, jakie jest znaczenie kolorów porządkowanych „sopli" i jakie znaczenie mają kolory kresek i kropek pojawiających się nad porządkowanymi i uporządkowanymi elementami (soplami).

Opis algorytmu porządkowania przez wybór

Przedstawmy teraz ścisły opis algorytmu porządkowania przez wybór, znanego jako **Selction Sort**.



Rysunek 9.
Demonstracja działania algorytmu porządkowania przez wybór w programie **Sortowanie**

Algorytm porządkowani przez wybór – Selection Sort

Dane: Liczba naturalna n i ciąg n liczb $x_1, x_2, \ldots, x_n$.
Wynik: Uporządkowanie danego ciągu liczb od najmniejszej do największej, czyli ciąg wynikowy spełnia nierówności $x_1 \le x_2 \le \ldots \le x_n$. (*Uwaga.* Elementy ciągu danego i wynikowego oznaczamy tak samo, gdyż porządkowanie odbywa się „w tym samym miejscu".)
Krok 1. Dla $i = 1, 2, \ldots, n-1$ wykonaj kroki 2 i 3, a następnie zakończ algorytm.
Krok 2. Znajdź k takie, że x_k jest najmniejszym elementem w ciągu $x_i, \ldots, x_n$.
Krok 3. Zamień miejscami elementy x_i oraz x_k.

Uwaga. Zauważ, że liczba k znaleziona w Kroku 2 może być równa i w tym kroku, a zatem w Kroku 3 ten sam element jest zamieniany miejscami ze sobą. Z taką sytuacja mamy do czynienia w piątej iteracji przykładowej demonstracji działania algorytmu, przedstawionej na rys. 7, gdzie element 4 jest zamieniany ze sobą.

Ćwiczenie 22. Można porządkować nie tylko liczby, ale również litery oraz inne znaki. Zastosuj algorytm porządkowania przez wybór do ustawienia ciągu liter MATERIAŁYDOZAJĘĆZINFORMATYKI w porządku alfabetycznym.

Ćwiczenie 23. Zapisz algorytm **Selection Sort** w postaci schematu blokowego. Zauważ, że Krok 2 tego algorytmu polega na znalezieniu indeksu elementu najmniejszego w podciągu jeszcze nieuporządkowanym. Możesz w tym celu skorzystać ze schematu blokowego algorytmu **Min**, odpowiednio go dostosowując do potrzeb algorytmu **Selection Sort**.

Komputerowa realizacja algorytmu Selection Sort

Z opisu algorytmu **Selection Sort** wynika, wspomnieliśmy już o tym wielokrotnie, że jest on iteracją innego algorytmu, algorytmu znajdowania najmniejszego elementu, zastosowanego do coraz krótszego podciągu danych. W takim przypadku, gdy jeden algorytm korzysta z wyników innego algorytmu, dobrze jest wydzielić ten pod algorytm jako niezależny podprogram. Uczynimy to teraz tutaj.

Wydzielone podprogramy nazywają się powszechnie **procedurami**. Czasem mogą one przyjmować postać **funkcji niestandardowej**. Na ogół w takich podprogramach można określić parametry, które służą do komunikacji, czyli do przekazywania pewnych wartości, między podprogramem a programem głównym.

Przypuśćmy, że chcemy Krok 2 algorytmu zapisać w postaci procedury o nazwie `IndexMin`. Danymi w tym przypadku jest ciąg danych, zapisany w tablicy `x`, oraz miejsce `i`, od którego ten podprogram ma szukać elementu najmniejszego, a wynikiem – indeks tego elementu najmniejszego. Taka procedura może mieć postać:

```
procedure IndexMin(x:tablicax; i:integer; var k:integer);
 var imin,j,min:integer;
begin
 min:=x[i]; imin:=i;
 for j:=i+1 to n do
  if min>x[j] then begin min:=x[j]; imin:=j end;
 k:=imin
end;
```

Teraz, realizacja Kroków 1 – 3 z algorytmu **Selection Sort** może mieć następującą postać:

```
 for i:=1 to n-1 do begin
  IndexMin(x,i,k);
  y:=x[i]; x[i]:=x[k]; x[k]:=y
 end
```

A cały program może przyjąć postać:

```
Program SelectioSort;
 const m=100;      {100 jest maksymalną długością danych}
 type tablicax=array[1..m] of integer; {deklaracja typu tablicy}
 var i,k,n,y:integer;
     x:tablicax;    {deklaracja tablicy}
 procedure IndexMin(x:tablicax; i:integer; var k:integer);
  var imin,j,min:integer;
 begin
  min:=x[i]; imin:=i;
  for j:=i+1 to n do
   if min>x[j] then begin min:=x[j]; imin:=j end;
  k:=imin
 end;
begin
 read(n);
 for i:=1 to n do read(x[i]);
 for i:=1 to n-1 do begin
  IndexMin(x,i,k);
  y:=x[i]; x[i]:=x[k]; x[k]:=y
 end;
 for i:=1 to n do writeln(x[i])
end.
```

Ćwiczenie 24. Uruchom powyższy program `SelctionSort` na komputerze i przetestuj jego działanie na kilku przykładach ciągów o różnych długościach – Twój program powinien działać poprawnie dla wszystkich ciągów

o długościach od 1 do 100. Zastosuj otrzymany program do losowych danych, a także do danych uporządkowanych i odwrotnie uporządkowanych.

W implementacji algorytmu **Selection Sort**, zamiast procedury można użyć **funkcji niestandardowej**, której wartością będzie poszukiwany indeks najmniejszego elementu w podciągu, który nie został jeszcze uporządkowany. W definicji funkcji, zbędny będzie parametr k, bo jego wartość przypiszemy nazwie funkcji, musimy jedynie zadeklarować na końcu pierwszego wiersza tekstu funkcji, jakiego typu będzie to wartość. Oto ta funkcja – nazwiemy ją tak samo jak procedurę:

```
function IndexMin(x:tablicax; i:integer):integer;
 var imin,j,min:integer;
begin
 min:=x[i]; imin:=i;
 for j:=i+1 to n do
  if min>x[j] then begin min:=x[j]; imin:=j end;
 IndexMin:=imin
end;
```

Ćwiczenie 25. W powyższym programie `SelctionSort` zamień procedurę `IndexMin` na funkcję `IndexMin` i dokonaj odpowiednich zmian w programie. Uruchom zmodyfikowany program i przetestuj poprawność jego działania.

Z programu `SelectioSort` można również wydzielić procedurę, będącą pełną realizacją algorytmu **Selection Sort**, by móc jej użyć jako podprogramu sortującego w innych programach.

Ćwiczenie 26. Sporządź opis procedury `SelctionSort`, będącej realizacją algorytmu **Selection Sort**. Przyjmij, że w nagłówku ta procedura ma następującą postać:

```
procedure SelectionSort(n:integer; var x:tablicax);
```

Następnie napisz program, w którym umieścisz tę procedurę i użyjesz go do jej testowania. W części początkowej (nagłówkowej) tego programu powinny się znaleźć definicje stałych i typów oraz deklaracje zmiennych i procedur używanych w programie, a w części głównej programu (czyli w bloku programu) – wczytywanie danych, wywołanie procedury sortującej i wyprowadzenie wyników.

Złożoność algorytmu Selection Sort

Obliczmy teraz, ile porównań i zamian elementów, w zależności od liczby elementów w ciągu n, jest wykonywanych w algorytmie **Selection Sort** oraz w jego komputerowych implementacjach. W tym celu wystarczy zauważyć, o czym pisaliśmy już powyżej, że algorytm jest iteracją algorytmu znajdowania najmniejszego elementu w ciągu, a ciąg, w którym szukamy najmniejszego elementu, jest w kolejnych iteracjach coraz krótszy. Liczba przestawień elementów jest równa liczbie iteracji, gdyż elementy są przestawiane jedynie na końcu każdej iteracji, których jest $n - 1$, a więc wynosi $n - 1$. Jeśli zaś chodzi o liczbę porównań, to wiemy już, że algorytm znajdowania minimum w ciągu wykonuje o jedno porównanie mniej niż jest elementów w ciągu. Ponieważ w każdym kroku liczba elementów w przeszukiwanym podciągu jest o jeden mniejsza, cały algorytm porządkowania przez wybór, dla ciągu danych złożonego na początku z n elementów wykonuje liczbę porównań równą:

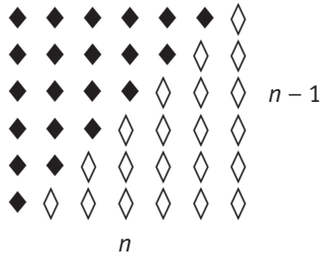
$$(n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

Wartość tej sumy można obliczyć wieloma sposobami. Przedstawimy dwa z nich – są one ciekawe przez swoją prostotę. Inny sposób, w którym korzysta się ze wzoru na sumę postępu arytmetycznego, pomijamy tutaj, jako oczywisty dla tych, którzy wiedzą, co to jest postęp arytmetyczny – nasze sposoby tego nie wymagają.

Dowód geometryczny

Kolejne liczby naturalne od 1 do $n - 1$ można przedstawić w postaci trójkąta, którego wiersz i, licząc od dołu, zawiera i diamentów (na rys. 10 są to czarne diamenty). Dwa takie same trójkąty pasują do siebie i tworzą prostokąt zawierający $(n - 1)n$ diamentów, zatem wartość powyższej sumy jest połową liczby wszystkich diamentów w całym prostokącie, czyli jest równa:

$$\frac{(n-1)n}{2}$$



Ten geometryczny dowód, zamieszczony obok, znali już starożytni Grecy. Na podstawie geometrycznej interpretacji, wartość sumy kolejnych liczb naturalnych nazywali **liczbami trójkątnymi.**

Rysunek 10.
Ilustracja geometrycznego wyznaczania wartości sumy kolejnych liczb naturalnych od 1 do $n - 1$

Spostrzeżenie nudzącego się geniusza
Anegdota mówi, że nauczyciel matematyki w klasie, do której uczęszczał młody Carl Friedrich Gauss (1777-1855), jeden z największych matematyków w historii, by zająć przez dłuższy czas swoich uczniów żmudnymi rachunkami, dał im do obliczenia wartość sumy stu początkowych liczb naturalnych, czyli 1 + 2 + 3 + ... + 98 + 99 + 100. Nie cieszył się jednak zbyt długo spokojem, po chwili otrzymał bowiem gotową odpowiedź od Carla, który szybko zauważył, że suma liczb w skrajnych parach, 1+100, 2+99, 3+98 itd. aż do 50+51 jest taka sama, a takich par jest połowa ze stu, czyli z liczby wszystkich elementów. Stąd natychmiast otrzymał powyższy wzór[8].

Otrzymaliśmy wzór na liczbę porównań w algorytmie porządkowania przez wybór, który zależy tylko od liczby porządkowanych elementów. Można uznać za słabą stronę tego algorytmu, że wykonuje on taką samą liczbę działań (porównań i przestawień) na ciągach o tej samej długości, bez względu na stopień ich uporządkowania. W następnym podpunkcie wspominamy o metodzie porządkowania, która jest „bardzo czuła" na stopień uporządkowania elementów w porządkowanym ciągu.

Ćwiczenie 27. Skorzystaj z programu **Sortowanie** i wykonaj kilka razy demonstrację sortowania przez wybór dla tej samej liczby elementów, np. 75 i dla różnych typów danych: przypadkowych, czyli losowych (wielokrotnie) i dla danych posortowanych i odwrotnie posortowanych. Za każdym razem zanotuj liczbę wykonanych porównań. Co zaobserwowałeś?

[8] Przekonaj się, że w rozumowaniu Carla Gaussa nie ma błędu – jest ono poprawne bez względu na to, czy ilość sumowanych liczb jest parzysta czy nieparzysta.

Inne algorytmy porządkowania
Znanych jest bardzo wiele algorytmów porządkowania liczba i innych obiektów przechowywanych w komputerze, jak słów, dat (to nie są liczby tylko układy liczb), rekordów (czyli układów danych różnych typów). Temu zagadnieniu poświęcono wiele opasłych książek, np. Donald Knuth napisał na ten temat ponad 1000 stron jeszcze w latach 60. XX wieku, patrz [6]. Niektóre z tych algorytmów będą jeszcze rozważane na tych zajęciach. W programie **Sortowanie** można zapoznać się z demonstracją najbardziej popularnych algorytmów, np. z **algorytmem bąbelkowym**, o którym nie będziemy więcej mówić na tych zajęciach, jedynie poniżej.

Ćwiczenie 28. W sortowaniu liczb metodą bąbelkową wykorzystuje się spostrzeżenie, że jeśli ciąg nie jest jeszcze uporządkowany, to istnieje para sąsiednich liczb, które znajdują się w złej kolejności, a więc należy je przestawić. **Algorytm bąbelkowy** jest uporządkowaną realizacją sprawdzania, czy istnieją takie pary, i ich zamian. Zapoznaj się z demonstracją algorytmu bąbelkowego w programie **Sortowanie**. Napisz program, który jest implementacją tego algorytmu.

Ćwiczenie 29. Zauważ ciekawą własność demonstrowanej w programie **Sortowanie** implementacji algorytmu bąbelkowego – jeśli ciąg jest uporządkowany, to algorytm ten przebiega tylko raz przez ciąg, a więc wykonuj bardzo mało operacji. Faktycznie jest to najszybciej działający algorytm w przypadku, gdy sortowany ciąg jest uporządkowany. Przejdź do opcji porównania algorytmów i do porównania wybierz sortowanie przez wybór i sortowanie bąbelkowe, a jako ciąg porządkowany wybierz dane posortowane. Czy w swoim programie z poprzedniego ćwiczenia postępujesz tak, aby w przypadku danych posortowanych algorytm tylko raz przebiegał przez ciąg?

6 POSZUKIWANIE INFORMACJI W ZBIORZE

Komputery bardzo ułatwiają szybkie poszukiwanie informacji umieszczonych na płytach CD i na serwerach sieci Internet. Jest to zasługą nie tylko szybkości działania procesorów, ale też dobrej organizacji pracy, dzięki czemu pozostaje im ... niewiele do roboty. Przypomnijmy poczynione założenia, dotyczące przeszukiwanych zbiorów. Zbiór może zawierać elementy powtarzające się (czyli o takich samych wartościach) i w obliczeniach jest dany w postaci ciągu, który na ogół jest poprzedzony liczbą jego elementów. Ciąg ten może być uporządkowany, np. od najmniejszego do największego elementu, lub nieuporządkowany.

W pierwszym rozdziale zajmowaliśmy się znajdowaniem szczególnych elementów w zbiorze nieuporządkowanym, najmniejszego i największego. Teraz będziemy rozważać problem poszukiwania w ogólniejszej postaci.

Problem poszukiwania elementu w zbiorze

Dane: Zbiór elementów w postaci ciągu n liczb $x_1, x_2, ..., x_n$.
Wyróżniony element y.
Wynik: Jeśli y należy do tego zbioru, to podaj jego miejsce (indeks) w ciągu, a w przeciwnym razie – sygnalizuj brak takiego elementu w zbiorze.

Problem poszukiwania ma bardzo wiele zastosowań i jest rozwiązywany przez komputer na przykład wtedy, gdy w jakimś ustalonym zbiorze informacji staramy się znaleźć konkretną informację. Rozważana przez nas wersja tego problemu jest bardzo prosta, na ogół bowiem zbiory i ich elementy mają bardzo złożoną postać, nie są ograniczone tylko do pojedynczych liczb. Przedstawione przez nas metody mogą być jednak uogólnione na bardziej złożone sytuacje problemowe.

W następnych podpunktach, najpierw rozwiązujemy problem poszukiwania w dowolnym zbiorze elementów, a później – w zbiorze uporządkowanym.

6.1 POSZUKIWANIE ELEMENTU W ZBIORZE NIEUPORZĄDKOWANYM

Jeśli nic nie wiemy o elementach w ciągu danych $x_1, x_2, ..., x_n$, to aby stwierdzić, czy wśród nich jest element równy danemu y, musimy sprawdzić każdy z elementów tego ciągu, gdyż element y może się znajdować w dowolnym miejscu ciągu, a w szczególnym przypadku może go tam nie być. W takim przypadku stosujemy **przeszukiwanie** (lub **poszukiwanie**) **liniowe**, które stosowaliśmy w rozdz. 1 do znajdowania w ciągu elementu najmniejszego lub największego. Na ogół takie przeszukiwanie odbywa się „od lewej do prawej", czyli od początku do końca ciągu. Można je opisać następująco.

Algorytm poszukiwania liniowego

Dane: Zbiór elementów w postaci ciągu n liczb $x_1, x_2, ..., x_n$.
Wyróżniony element y.
Wynik: Jeśli y należy do tego zbioru, to podaj jego miejsce (indeks) w ciągu, a w przeciwnym razie – sygnalizuj brak takiego elementu w zbiorze.
Krok 1. Dla $i = 1, 2, ..., n$, jeśli $x_i = y$, to przejdź do kroku 3.
Krok 2. Komunikat: W ciągu danych nie ma elementu równego y.
Zakończ algorytm.
Krok 3. Element równy y znajduje się na miejscu i w ciągu danych.
Zakończ algorytm.

Ćwiczenie 30. Utwórz schemat blokowy dla powyższego algorytm, a następnie zapisz go w postaci programu komputerowego.

Jeśli element y znajduje się w przeszukiwanym ciągu, to algorytm kończy działanie po natknięciu się na niego po raz pierwszy, a jeśli nie ma go w tym ciągu to kończy się po dojściu do końca ciągu. W obu przypadkach liczba działań jest proporcjonalna do liczby elementów w ciągu. W pierwszym przypadku najwięcej operacji jest wykonywanych wówczas, gdy poszukiwany element jest na końcu ciągu.

Algorytm poszukiwania może wykonywać różną liczbę iteracji nawet dla ustalonego ciągu danych, zależy ona bowiem od wartości poszukiwanego elementu y. Mamy więc okazję, by wprowadzić nową instrukcję iteracyjną, w której liczba iteracji może zależeć od spełnienia podanego warunku. W przypadku algorytmu poszukiwania, kończy on działanie w jednej z dwóch sytuacji: albo został znaleziony poszukiwany element y, albo został przejrzany cały ciąg i nie znaleziono tego elementu. Musimy uwzględnić jedną i drugą ewentualność. Umożliwia nam to następująca funkcja – przyjmujemy, że wartością funkcji jest indeks znalezionego elementu, jeśli znajduje się on w ciągu, lub – 1, jeśli element o wartości y nie istnieje w ciagu:

```
function PrzeszukiwanieLiniowe(n,y:integer; x:tablicax):integer;
 {Wartoscia funkcji jest indeks elementu tablicy
  rownego y, lub -1, jesli brak takiego elementu w ciągu.}
  var i:integer;
begin
 i:=1;
 while (x[i]<>y) and (i<n) do i:=i+1;
 if x[i]=y then PrzeszukiwanieLiniowe:=i
 else PrzeszukiwanieLiniowe:=-1
end; {PrzeszukiwanieLiniowe}
```

Warunkowa instrukcja iteracyjna:

```
 while (x[i]<>y) and (i<n) do i:=i+1;
```

jest wykonywana tak długo, jak długo spełniony jest warunek:

```
 (x[i]<>y) and (i<n)
```

Czyli, gdy badany element ciągu jest różny od *y* oraz nie został jeszcze osiągnięty koniec ciągu. Wtedy zwiększany jest bieżący indeks elementów ciągu. Po tej instrukcji następuje złożona instrukcja warunkowa:

```
 if x[i]=y then PrzeszukiwanieLiniowe:=i
 else PrzeszukiwanieLiniowe:=-1
```

której zadaniem jest zbadanie, z jakiego powodu nastąpiło zakończenie iteracji. Jeśli `x[i]=y`, to w ciągu został znaleziony element równy *y*, a w przeciwnym razie (`else`) nie ma w ciągu elementu o wartości *y*.

Ćwiczenie 31. Zmodyfikuj ewentualnie swoją implementację algorytmu przeszukiwania liniowego po zapoznaniu się z uwagami powyżej i przetestuj jej poprawność na odpowiedniej liczbie przykładów. Dla ustalonego ciągu, za *y* przyjmij kolejne wartości elementów tego ciągu, a także liczby, które są: mniejsze od najmniejszego elementu ciągu, większe od największego elementu ciągu, leżą pomiędzy wartościami elementów ciągu.

Przeszukiwanie liniowe z wartownikiem

Ciekawe własności ma niewielka modyfikacja powyższego algorytmu, wykorzystująca specjalny element, umieszczony na końcu ciągu, zwany **wartownikiem**. Rolą wartownika jest „pilnowanie", by proces przeszukiwania nie wyszedł poza ciąg. Jak wiemy, gdy ciąg zawiera element o wartości *y*, to przeszukiwanie kończy się na tym elemencie. Aby mieć pewność, że przeszukiwanie zawsze zakończy się na elemencie o wartości *y*, dołączamy na końcu ciągu element – wartownika – właśnie o wartości *y*. W efekcie, przeszukiwanie zawsze zakończy się znalezieniem elementu o wartości *y*, należy jedynie sprawdzić, czy znaleziony element *y* znajduje się na dołączonej pozycji zbioru, czy też wystąpił wcześniej. W pierwszym przypadku, badany zbiór nie zawiera elementu równego *y*, a w drugim – *y* należy do zbioru. Widać stąd, że dołączony do zbioru element odgrywa rolę jego wartownika – nie musimy bowiem sprawdzać, czy przeglądanie objęło cały zbiór czy nie – zawsze zatrzyma się ono na szukanym elemencie, którym może być dołączony właśnie element.

A oto fragment przeszukiwania z wartownikiem:

```
begin
 i:=1;
 x[n+1]:=y;
 while x[i]<>y do i:=i+1;
 if i<=n then PrzeszukiwanieLinioweWartownik:=i
 else PrzeszukiwanieLinioweWartownik:=-1
end;
```

Ćwiczenie 32. Zmodyfikuj swój program otrzymany w poprzednim ćwiczeniu, włączając do niego pomysł z wartownikiem. Wszechstronnie przetestuj zmodyfikowany program, czy rzeczywiście poprawnie rozwiązuje problem poszukiwania. Podobnie, jak w ćwicz. 31, dla ustalonego ciągu,

za *y* przyjmij kolejne wartości elementów tego ciągu, a także liczby, które są: mniejsze od najmniejszego elementu ciągu, większe od największego elementu ciągu, leżą pomiędzy wartościami elementów ciągu.

6.2 POSZUKIWANIE ELEMENTU W ZBIORZE UPORZĄDKOWANYM

W tym podrozdziale zakładać będziemy, że poszukiwania elementów (informacji) są prowadzone w uporządkowanych zbiorach (ciągach) elementów – chcemy albo znaleźć element, albo umieścić go w takim zbiorze z zachowaniem uporządkowania.

Porządek w informacjach

Zbiory mogą mieć różną strukturę – mogą to być książki w bibliotece, hasła w encyklopedii, liczba w ustalonym przedziale lub numery w książce telefonicznej. Te przykłady są bliskie codziennym sytuacjom, w których należy odszukać pewną informację i zapewne stosowane przez Was w tych przypadkach metody są podobne do opisanych tutaj. Naszymi rozważaniami chcemy utwierdzić Was w przekonaniu, że:

integralną częścią informacji jest jej uporządkowanie,

gdyż w przeciwnym razie ... nie jest to informacja. To stwierdzenie nie jest naukowym określeniem informacji[9], ale odnosi się do informacji w potocznym znaczeniu, do informacji, które nas zalewają i nieraz przytłaczają, do informacji, wśród których mamy odnaleźć tę nam potrzebną lub „zrobić wśród nich porządek". Podstawowym przygotowaniem do życia w erze i społeczeństwie informacji jest bowiem nabycie umiejętności takiego postępowania z informacją (uporządkowaną oczywiście), by w posługiwaniu się nią korzystać z jej uporządkowania, nie psuć go i ewentualnie naprawiać, gdy ulega zniszczeniu, lub gdy informacja się rozrasta.

Wykonaj teraz ćwiczenie, które zapewne wykonywałeś już nieraz w swoim życiu, nie zdając sobie nawet z tego sprawy.

[9] W teorii informacji, informacja jest definiowana jako „miara niepewności zajścia pewnego zdarzenia spośród skończonego zbioru zdarzeń możliwych" — na podstawie *Nowej encyklopedii powszechnej PWN* .

Ćwiczenie 33. Weź do ręki jedną z książek: słownik ortograficzny, słownik polsko-angielski lub książkę telefoniczną, wybierz trzy słowa zaczynające się na litery: c, l oraz w i znajdź je w wybranej książce. Zanotuj, ile razy ją otwierałaś, zanim znalazłeś kartkę z poszukiwanym słowem.

Jeśli książka, którą wybrałeś, ma między 1000 a 2000 stron, to dla znalezienia jednego słowa nie powinieneś otwierać jej częściej niż 11 razy; jeśli ma między 500 a 1000 stron – to nie częściej niż 10 razy; jeśli między 250 a 500 stron – to nie częściej niż 9 razy itp.

Skąd to wiemy? Przypuszczamy, że w poszukiwaniu hasła, po zajrzeniu na wybraną stronę wiesz, że znajduje się ono przed nią, albo po niej, możesz więc jedną z części książki pominąć w dalszych poszukiwaniach. Co więcej, w nieodrzuconej części kartek wybierasz jako kolejną tę, która jest bliska środka, lub leży w pobliżu litery, na którą zaczyna się poszukiwany wyraz. Stosujesz więc – może nawet o tym nie wiedząc – metodę poszukiwania, która polega na **podziale (połowieniu) przeszukiwanego zbioru**. Możesz ją zastosować, bo przeszukiwany zbiór jest uporządkowany. A ile prób musiałbyś wykonać, gdyby hasła w słowniku nie były uporządkowane?

Porównaj teraz:

- W alfabetycznym spisie telefonów na 1000 stronach wystarczy przejrzeć co najwyżej 10 kartek, by znaleźć numer telefonu danej osoby.
- Gdyby nazwiska abonentów nie były ustawione alfabetycznie, to w najgorszym przypadku musiałbyś przejrzeć je wszystkie 1000!

Pamiętaj. Staraj się wykorzystywać uporządkowanie zbioru, który przeszukujesz – znacznie skraca to Twoją pracę. Utrzymuj również porządek w swoich zbiorach (rzeczach), a będziesz mniej czasu tracił na ich przeszukiwanie.

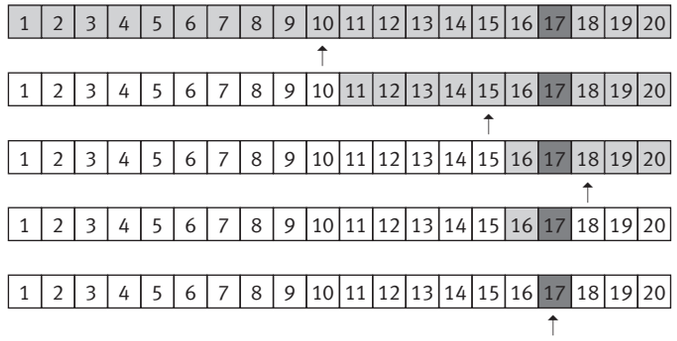
Czy to porównanie nie świadczy o potędze uporządkowania i o sile algorytmu zastosowanego do uporządkowanego wykazu?

Zabawa w zgadywanie liczb

Strategię podobną do poszukiwania w alfabetycznych spisach stosuje się podczas gry, polegającej na zgadywania ukrytej przez drugą osobę liczby.

Ćwiczenie 34. Wybierz sobie Partnera do gry, która polega na odgadywaniu liczby naturalnej wybranej z przedziału [1, *n*]. Partner wybiera liczbę, a Ty masz ją odgadnąć. Na Twój wybór Partner może jedynie odpowiedzieć: „tak", „za mała" lub „za duża". Jaką przyjmiesz strategię odgadywania liczby, by ją znaleźć w możliwie najmniejszej liczbie prób?

Jeśli nie od razu, to na pewno po kilku próbach odkryjesz, że najlepsza strategia polega na podawaniu środkowych liczb z przedziału, w którym znajduje się poszukiwana liczba. Przypuśćmy, że poszukujemy liczby w przedziale [1, 100]. Jeśli pierwszym wyborem byłaby liczba 75 i otrzymalibyśmy odpowiedź „za duża", to pozostałby do przeszukania przedział [1, 75]. Jeśli natomiast wybierzemy w pierwszej próbie 50, to bez względu na to, jaką liczbę wybrał Partner, pozostanie do przeszukania nie więcej niż pięćdziesiąt liczb, w przedziale [1, 49] albo [51, 100].



Rysunek 11.
Przykład zastosowania binarnej metody przeszukiwania do znalezienia liczby 17 w ciągu złożonym z dwudziestu początkowych liczb naturalnych. Podciemniono podciąg, w którym znajduje się poszukiwana liczba, a strzałka wskazuje wybraną przez algorytm w nim liczbę

Na rys. 11 są przedstawione kolejne kroki przeszukiwania, prowadzące do znalezienia liczby 17 w przedziale liczb [1, 20]. W każdym kroku jest wybierana środkowa liczba spośród pozostałych – ta liczba jest wskazana strzałką poniżej niej. Jeśli w ciągu pozostała parzysta ilość liczb, to spośród dwóch środkowych zawsze jest wybierana wcześniejsza. Podciąg liczb, który pozostał do przeszukania, jest oznaczony na czerwono (na szarym tle), by zwrócić Twoją uwagę na fakt, że w każdym kroku odgadywania ten ciąg jest redukowany przynajmniej o połowę.

Przedstawione przykłady poszukiwania przez połowienie w zbiorze uporządkowanym ilustrują, że ta metoda jest kolejnym przykładem **zasady dziel i zwyciężaj**.

Algorytm poszukiwania przez połowienie

Algorytm poszukiwania przez połowienie jest zwany również **binarnym poszukiwaniem**. Opiszemy teraz ten algorytm dla trochę ogólniejszej sytuacji niż w zabawie w odgadywanie liczb. Po pierwsze zauważmy, że w podanej wyżej metodzie nie mają znaczenia wartości elementów w tablicy, tak długo, jak długo są uporządkowane. Musimy mieć jedynie pewność, że po porównaniu wskazanej w tablicy liczby z poszukiwaną i po wybraniu połowy zbioru, poszukiwana liczba znajduje się w wybranej połowie, a do tego wystarczy, by elementy w tablicy były uporządkowane, nie muszą być koniecznie kolejnymi liczbami. Tablica może również zawierać takie same liczby – wtedy oczywiście zajmują one miejsca obok siebie. Po drugie, poszukiwana liczba nie musi znajdować się w tablicy – wtedy naszą odpowiedzią będzie jakaś specjalnie wybrana liczba, np. –1.

Przyjmijmy, że przeszukiwany ciąg liczb jest umieszczony w tablicy $x[k..l]$. Załóżmy dodatkowo, że wartość poszukiwanego elementu y mieści się w przedziale wartości elementów w tej tablicy, czyli $x_k \le y \le x_l$. Algorytm, który podajemy gwarantuje, że w trakcie jego działania, podobnie jak w grze w odgadywanie liczby, przeszukiwany przedział zawiera element y, czyli $x_{lewy} \le y \le x_{prawy}$. Ta własność oraz to, że długość tego przedziału zmniejsza się w każdej iteracji (zob. krok 3), zapewniają, że poniższy algorytm jest poprawny.

Algorytm poszukiwania przez połowienie (algorytm binarnego przeszukiwania)

Dane: Uporządkowany ciąg liczb w tablicy $x[k..l]$, tzn. $x_k \le x_{k+1} \le \ldots \le x_l$; oraz element y spełniający nierówności $x_k \le y \le x_l$

Wyniki: Takie s ($k \le s \le l$), że $x_s = y$, lub przyjąć $s = -1$, jeśli $y \neq x_i$ dla każdego i ($k \le i \le l$).

Krok 1. *lewy* := *k*; *prawy* := *l*; {Początkowe końce przeszukiwanego przedziału.}

Krok 2. Jeśli *lewy* › *prawy*, to przypisz $s := -1$ i zakończ algorytm. {Oznacza to, że poszukiwanego elementu *y* nie ma w przeszukiwanej tablicy.}

Krok 3. *s*:=(*lewy* + *prawy*) div 2; {Operacja div oznacza dzielenie całkowite.}
Jeśli $x_s = y$, to zakończ algorytm. {Znaleziono element *y* w przeszukiwanej tablicy.}
Jeśli $x_s < y$, to *lewy* := *s* + 1, a w przeciwnym razie *prawy* := *s* – 1.
Wróć do kroku 2.

Ćwiczenie 35. Zastosujmy wspólnie (sprawdzaj obliczenia) powyższy algorytm do przeszukania ciągu liczb znajdującego się w tablicy na rys. 11, ale będziemy szukać elementu $y = 17.5$, którego w niej nie ma. W pierwszej iteracji kroków 2 i 3 otrzymujemy $s := 10$ oraz *lewy* := 11, w drugiej iteracji: $s := 15$ i *lewy* := 16, w trzeciej: $s := 18$ i *prawy* := 17, w czwartej: $s := 16$ i *lewy* := 17, a w piątej: $s := 17$ i *lewy* := 18. W następnej iteracji, ponieważ *lewy* › *prawy*, następuje zakończenie algorytmu z wynikiem $s = -1$, oznaczającym, że nie znaleziono w tablicy elementu o wartości 17.5.

Implementacja poszukiwania przez połowienie

Jak podaje Donald E. Knuth [6], napisanie w pełni poprawnej komputerowej implementacji algorytmu poszukiwania przez połowienie, sprawiło kłopot wielu programistom. Wykonaj samodzielnie następne ćwiczenie, a jeśli będziesz miał z tym kłopoty lub będziesz chciał upewnić się, czy dobrze zrobiłeś, możesz zajrzeć do naszej implementacji, którą zamieszczamy poniżej.

Ćwiczenie 36. Napisz implementację algorytmu poszukiwania przez połowienie w postaci funkcji niestandardowej, której wartością jest *s* – indeks elementu równego *y*, jeśli taki element istnieje, lub – 1, jeśli takiego elementu nie ma. Wszechstronnie przetestuj swoją funkcję na ustalonym ciągu danych i wartościach y, przebiegających wszystkie wartości elementów z ciągu oraz wartości między nimi.

A oto nasza implementacja algorytmu przeszukiwania przez połowienie.

```
function PrzeszukiwanieBinarne(x:tablicax; k,l:integer;
                                  y:integer):integer;
  {Przeszukiwanie binarne ciagu x[k..l] w poszukiwaniu
   elementu y. Wartoscia funkcji jest indeks elementu
   tablicy rownego y, lub -1, jesli brak takiego elementu.
   W programie glownym nalezy zdefiniowac typ
   danych:  tablicax=array[1..n] of integer.}
 var Lewy,Prawy,Srodek:integer;
begin
 Lewy:=k;  Prawy:=l;
 while Lewy<=Prawy do begin
  Srodek:=(Lewy+Prawy) div 2;
  if x[Srodek]=y then begin
   PrzeszukiwanieBinarne:=Srodek; exit
  end;  {Element y nalezy do przeszukiwanego ciagu.}
  if x[Srodek]<y then Lewy:=Srodek+1
  else Prawy:=Srodek-1
 end;
 PrzeszukiwanieBinarne:=-1
end; {PrzeszukiwanieBinarne}
```

Złożoność algorytmu binarnego przeszukiwania

Nasuwa się teraz pytanie, ile porównań jest wykonywanych w algorytmie binarnego przeszukiwania. Załóżmy, że poszukiwany element znajduje się w ciągu – bo jeśli go tam nie ma, to jest wykonywana jedna dodatkowa iteracja (przekonaj się o tym). Dla dwudziestu liczb z przykładu na rys. 11, poszukiwaną liczbę znaleźliśmy po czterech porównaniach – piątego nie musieliśmy już wykonywać, gdyż pozostała dokładnie jedna liczba – poszukiwana.

Ćwiczenie 37. Przeprowadź następujący eksperyment obliczeniowy: za poszukiwaną liczbę wybieraj kolejne liczby z przedziału [1, 20] i zanotuj, ile wykonałeś porównań w algorytmie binarnego przeszukiwania, by ją odnaleźć.

Pytanie o liczbę porównań w powyższym algorytmie można sformułować następująco: ile razy należy odrzucać połowę bieżącego ciągu, by pozostał tylko jeden element (zauważmy tutaj, że jeśli elementu y nie ma w ciągu, to kontynuujemy algorytm aż do wyczerpania wszystkich elementów, czyli wykonujemy o jeden krok więcej). Jeśli $n = 32$, to jeden element pozostaje po pięciu podziałach, a jeśli $n = 16$ — to po czterech. Stąd można wywnioskować, że jeśli wartość n zawiera się między 16 a 32, to wykonujemy nie więcej niż pięć porównań. Wyniki ćwicz. 37 powinny to potwierdzić. Jak tę obserwację można uogólnić? Zapewne jest to związane z potęgą liczby 2, a dokładniej z najmniejszym wykładnikiem potęgi, której wartość nie jest mniejsza od n. Pojawia się więc tutaj w naturalny sposób funkcja odwrotna do potęgowania – **logarytm**. Można nawet przyjąć „informatyczną" definicję tej funkcji: $\log_2 n$ jest równy liczbie kroków prowadzących od n do 1, w których bieżąca liczba jest zastępowana przez zaokrąglenie w górę jej połowy.

Algorytm binarnego umieszczania

Algorytm binarnego przeszukiwania ma dość istotne uogólnienie, gdy dla elementu y, bez względu na to, czy należy do ciągu czy nie, chcemy znaleźć takie miejsce, by po wstawieniu go tam, ciąg pozostał uporządkowany. Odpowiedni algorytm można w tym przypadku nazwać **binarnym umieszczaniem**.

Poniższy algorytm, dla danego elementu y znajduje miejsce, w które można go wstawić z zachowaniem porządku. Zakładamy, że umieszczany element nie jest mniejszy od lewego końca przedziału, czyli $y \geq x_k$, co możemy sprawdzić przed uruchomieniem algorytmu i jeśli ten warunek nie jest spełniony, to wstawić y na początku ciągu.

Algorytm binarnego umieszczania

Dane: Uporządkowany ciąg liczb w tablicy x [$k..l$], tzn. $x_k \leq x_{k+1} \leq \ldots \leq x_l$; oraz element y spełniający nierówność $y \geq x_k$.

Wynik: Miejsce dla y w ciągu $x[k..l]$, tzn. największe r takie, że $x_r \leq y < x_{r+1}$, jeśli, lub $r = l$, gdy $x_l \leq y$. {Drugi przypadek odpowiada sytuacji, gdy wartość elementu y wykracza poza prawy koniec przedziału.}

Krok 1. *lewy* := k; *prawy* := l; {Początkowe końce przedziału przeszukiwań.}

Krok 2. s: = (*lewy* + *prawy*) / 2
Jeśli $x_s \leq y$, to *lewy* := s, a w przeciwnym razie *prawy* := $s - 1$.
Jeśli *lewy* = *prawy*, to zakończ algorytm – wtedy r = *lewy*, w przeciwnym razie powtórz krok 2.

Ćwiczenie 38. Wykonaj algorytm binarnego umieszczania dla danych z ćwicz. 35, czyli dla ciągu złożonego z dwudziestu pierwszych liczb naturalnych począwszy od 1 i dla $y = 17.5$. Porównaj na tym przykładzie działanie obu algorytmów binarnych, przeszukiwania i umieszczania.

Ćwiczenie 39. Sprawdź na przykładzie poprawność algorytmu binarnego umieszczania, gdy wartość elementu y wykracza poza prawy koniec przedziału, czyli gdy $y \geq x_l$. A jaki otrzymasz wynik, gdy wartość poszukiwanego elementu wykracza poza lewy koniec przedziału, czyli gdy $y < x_k$?

Ćwiczenie 40. Napisz implementację algorytmu binarnego umieszczania w postaci funkcji niestandardowej, której wartością jest r – indeks elementu ciągu, po którym należy umieścić w nim element y. Wszechstronnie przetestuj swoją funkcję na ustalonym ciągu danych i wartościach y, przebiegających zgodnie z sugestiami z ćwicz. 38 i 39.

Poszukiwanie interpolacyjne, czyli poszukiwania w słownikach

Czy rzeczywiście przeszukiwanie binarne jest najszybszą metodą znajdowania elementu w zbiorze uporządkowanym?

Wyobraźmy sobie, że mamy znaleźć w książce telefonicznej numer telefonu Pana Bogusza Alfreda. Wtedy zapewne skorzystamy z tego, że litera B jest blisko początku alfabetu i, owszem, zastosujemy metodę podziału. W pierwszej próbie nie będziemy jednak dzielić książki na dwie połowy, ale raczej spróbujemy trafić blisko tych stron, na których znajdują się nazwiska zaczynające się na literę B. W dalszych krokach będziemy postępować podobnie. Tę obserwację można wykorzystać w algorytmie poszukiwania. Zauważmy najpierw, że w algorytmach binarnych jest sprawdzana jedynie relacja, **czy** dana liczba y jest większa (lub mniejsza lub równa) od wybranej z ciągu, natomiast nie sprawdzamy i nie wykorzystujemy tego, **jak bardzo** jest większa. Podczas odnajdywania wyrazów w encyklopediach korzystamy natomiast z informacji, w jakim miejscu alfabetu znajduje się litera, którą rozpoczyna się poszukiwany wyraz, i w zależności od tego wybieramy odpowiednią porcję kartek. Strategia ta nazywa się **interpolacyjnym poszukiwaniem**, gdyż uwzględnia nie tylko położeniu szuka-

nej liczby względem środka ciągu, ale uwzględnia jej wartość względem rozpiętości krańcowych wartości w ciągu.

Szczegółowe informacje na temat interpolacyjnego poszukiwania można znaleźć w książce [9].

ZAKOŃCZENIE

Mamy nadzieję, że te zajęcia poszerzyły zrozumienie, na czym polega rozwiązywanie problemów w postaci algorytmicznej. Służyły temu przytoczone przykłady prostych problemów i wszechstronne ich omówienie wraz z implementacją w postaci programów komputerowych w wybranym języku programowania. Zachęcamy do dalszych zajęć w tym module tematycznym, które będą poświęcone innym problemom algorytmicznym oraz rozwijaniu umiejętności programowania.

LITERATURA

1. *I Olimpiada Informatyczna 1993/1994,* M.M. Sysło (red.), Wrocław, Warszawa 1994
2. *II Olimpiada Informatyczna 1994/1995,* M.M. Sysło (red.), Wrocław, Warszawa 1995
3. Cormen T.H., Leiserson C.E., Rivest R.L., *Wprowadzenie do algorytmów,* WNT, Warszawa 1997
4. Gurbiel E., Hard-Olejniczak G., Kołczyk E., Krupicka H., Sysło M.M., *Informatyka, Część 1 i 2, Podręcznik dla LO,* WSiP, Warszawa 2002-2003
5. Harel D., *Algorytmika. Rzecz o istocie informatyki,* WNT, Warszawa 1992
6. Knuth D.E., *Sztuka programowania, Tomy 1 – 3,* WNT, Warszawa 2003
7. Nievergelt J., Co to jest dydaktyka informatyki? *Komputer w Edukacji* 1/1994
8. Steinhaus H., *Kalejdoskop matematyczny,* WSiP, Warszawa 1989
9. Sysło M.M., *Algorytmy,* WSiP, Warszawa 1997
10. Sysło M.M., *Piramidy, szyszki i inne konstrukcje algorytmiczne,* WSiP, Warszawa 1998. Kolejne rozdziały tej książki są zamieszczone na stronie: www.wsipnet.pl/kluby/informatyka_ekstra.php?k=69
11. Wirth N., *Algorytmy + struktury danych = programy,* WNT, Warszawa 1980

W projekcie **Informatyka +**, poza wykładami i warsztatami,
przewidziano następujące działania:

- 24-godzinne kursy dla uczniów w ramach modułów tematycznych
- 24-godzinne kursy metodyczne dla nauczycieli, przygotowujące do pracy z uczniem zdolnym
- nagrania 60 wykładów informatycznych, prowadzonych przez wybitnych specjalistów i nauczycieli akademickich
- konkursy dla uczniów, trzy w ciągu roku
- udział uczniów w pracach kół naukowych
- udział uczniów w konferencjach naukowych
- obozy wypoczynkowo-naukowe.

Szczegółowe informacje znajdują się na stronie projektu **www.informatykaplus.edu.pl**

informatyka+

Algorytmika i programowanie

Bazy danych

Multimedia, grafika i technologie internetowe

Sieci komputerowe

Tendencje w rozwoju informatyki i jej zastosowań

Człowiek – najlepsza inwestycja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.