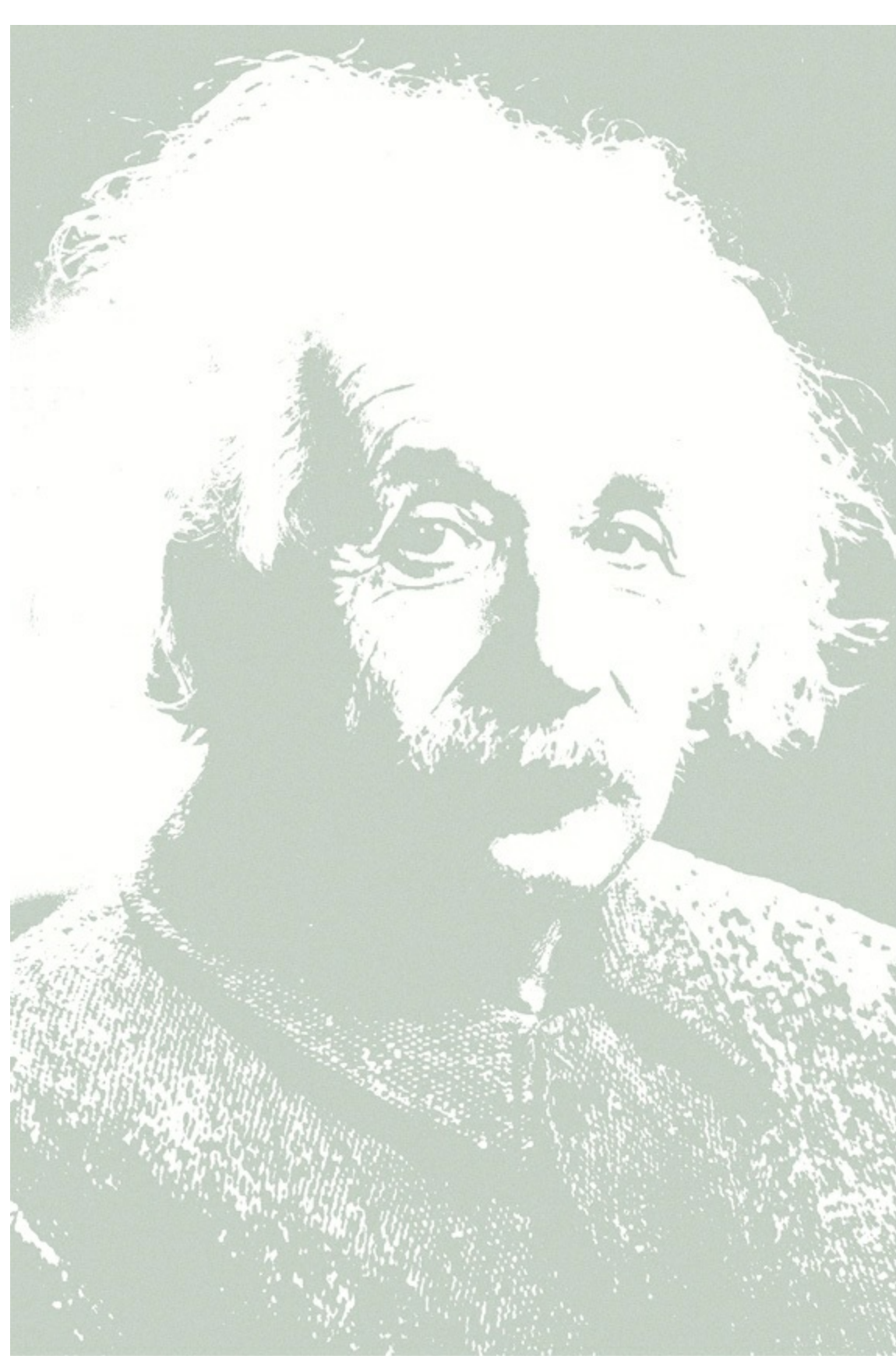


Z PROGRAMAMI
W JĘZYKACH
PASCAL I PYTHON

Maciej M. Sysło

ALGORYTMY





*Things should be as simple as possible,
but not simpler.*

***Wszystko powinno być tak proste,
jak to tylko możliwe, ale nie prostsze.***

Albert Einstein (1878 - 1955)

Od autora...

Książka została napisana z myślą o tych, których interesuje poznawanie sposobów tworzenia algorytmów oraz korzystanie z nich w rozwiązywaniu problemów. Zachęcamy do rozpoczęcia jej lektury od przeczytania w całości tej przedmowy, informacji o oznaczeniach i rozdziału 1.

Algorytm opisuje krok po kroku rozwiązanie postawionego problemu lub sposób osiągnięcia jakiegoś celu. Książka traktuje o tworzeniu algorytmów. Budujemy je i przedstawiamy, korzystając z metod informatycznych, aby mogły być wykonywane również przez komputer. Problemy zostały tak dobrane, aby wyjaśnić różne metody tworzenia rozwiązań, chociaż na ich wybór niewątpliwie wpływ miał również gust i zainteresowania autora. Książka jest więc nie tylko zbiorem algorytmów, choć można w niej znaleźć kompletne opisy wielu z nich. Najlepszą jednak drogą do ich poznania i zrozumienia działania jest prześledzenie, w jaki sposób one powstają. Ponadto, można zapoznać się z realizacją algorytmów w językach Pascal i Python.

...do uczniów

Książka może być pomocą w poznawaniu budowy i działania algorytmów przez wszechstronne prześledzenie procesu ich powstawania: od ścisłego opisu problemu, poprzez analizę różnych aspektów rozwiązania, po realizację w wybranej reprezentacji. Rozważania są ilustrowane przykładami i prostymi ćwiczeniami, które mają ułatwić wyjaśnienie kolejnych etapów tego procesu. Sprawdzianem rozumienia poznawanych algorytmów i zawartych w nich sposobów rozwiązywania problemów może być umiejętność ich wykorzystania w rozwiązaniach pokrewnych zadań, zamieszczonych w tekście i na końcu rozdziałów. Problemy zebrane na końcu książki są przeznaczone do samodzielnego rozwiązania, co będzie stanowić potwierdzenie nabycia poszerzonych umiejętności algorytmicznego myślenia.

Chociaż w tej książce algorytm służy przede wszystkim do przedstawienia rozwiązania problemu, obecnie jest rozumiany najczęściej jako przepis wykonania pewnego zadania przez komputer. Specyfika obliczeń komputerowych nakłada dodatkowe warunki zarówno na postać, w jakiej przedstawia się algorytmy, jak i przede wszystkim na ich własności, które mają gwarantować cechy rozwiązań wynikające z charakteru rozważanych problemów. Ma to wpływ na dobór sposobów opisu algorytmów, które z jednej strony powinny zapewniać ścisłość i jasność prezentacji, a z drugiej — ułatwiać ich analizowanie i tworzenie komputerowych realizacji.

...do nauczyciela

Z książki można korzystać bez względu na inne pomoce, w tym również informatyczne, stosowane w nauczaniu o algorytmach. Przykłady zadań prowadzące do algorytmów, intuicyjne wyprowadzanie algorytmów oraz ich słowne i graficzne reprezentacje mogą być przydatne na zajęciach prowadzonych z wykorzystaniem dowolnego systemu użytkowego lub systemu (języka) programowania.

Książka może być przydatna na lekcjach informatyki, matematyki, techniki lub innych przedmiotów, które dotyczą rozwiązywania problemów oraz posługiwania się przy tym komputerami.

...podziękowania

Wyrażam wdzięczność swoim współpracownikom z Instytutu Informatyki Uniwersytetu Wrocławskiego, z którymi przez wiele lat tworzyliśmy pomoce do nauczania informatyki, za dyskusje, sugestie i stwarzanie atmosfery do poszukiwań.

Książka powstała w czasie mojego pobytu na University of Oregon w Eugene (USA). Pobyt ten był sponsorowany przez Fulbright Senior Research Grant. Za stworzenie warunków do pracy dziękuję zarówno Fundacji Fulbrighta, jak i mojemu koledze z Uniwersytetu w Eugene, Andrzejowi Proskurowskiemu.

Składałam również podziękowania recenzentom, Iwonie Krajewskiej-Kranas i Witoldowi Kranasowi, za wiele cennych uwag, które pomogły ulepszyć prezentację.

At last but not at least, wyrażam wdzięczność Pani Redaktor Annie Łaskiej-Gmaj, której ta książka wiele zawdzięcza.

Wstęp do wydania Helion

W najnowszym wydaniu usunięto opisy algorytmów przygotowane z użyciem programu ELI oraz zostały dodane programy w języku Python.

Uzupełnieniem tej książki jest inna publikacja autora *Piramidy, szyszki i inne konstrukcje algorytmiczne*. Przedstawiono w niej sytuacje problemowe z różnych dziedzin i sposoby ich rozwiązywania w postaci algorytmicznej.

Książce towarzyszą zasoby elektroniczne na stronie <http://edukacja.helion.pl/algorytmika>. W szczególności znajdują się tam teksty programów w językach Pascal i Python, realizujących algorytmy opisane w tej książce.

Maciej M. Sysło
syslo@ii.uni.wroc.pl
syslo@mat.umk.pl
<http://mmsyslo.pl>

Wrocław, w lutym 2016 roku.

Wyróżnienia i oznaczenia w tekście

Tu dowiesz się:

- ▶ po co umieszczono punkty na początku każdego rozdziału — zawierają one informacje o treści danego rozdziału;
- ▶ jakie jest **znaczenie wyróżnień i oznaczeń** stosowanych w tej książce.


Wszystkie wyróżnienia w tekście, takie jak: różne rodzaje pisma (krojów i wielkości czcionek), kolor, umieszczanie fragmentów tekstu w wyróżnionych miejscach i w ramkach oraz ikony nie są tylko ozdobnikami, ale mają na celu przede wszystkim zwrócić uwagi na charakter prezentowanych informacji oraz powinny ułatwić korzystanie z tej książki. Pojęcia definiowane w tekście są pisane **pogrubioną czcionką** i zebrane na końcu książki w skorowidzu.

Czcionki pochylej użyto do zapisania symboli i wzorów matematycznych, nazw plików i folderów, linków oraz informacji ukazujących się na ekranie monitora, w tym również nazw klawiszy.

W takiej ramce zachodzącej na margines zamieszczamy informacje, które stanowią uzupełnienie zasadniczych rozważań: uwagi historyczne, ważniejsze zastosowania, uogólnienia rozpatrywanych zagadnień, a także różne ciekawostki. Miejsce umieszczenia tych uwag bynajmniej nie świadczy o ich marginalności.

Teksty programów w językach Pascal i Python, polecenia i komunikaty systemów oprogramowania są zapisane czcionką o stałej szerokości.

Wiele fragmentów tekstu poprzedzają ikony. Należą do nich rozważania, w których: omawiamy trudniejsze zagadnienia, dyskutujemy zagadnienia znajdujące się w innych opracowaniach lub zamieszczamy programy zapisane w

języku Pascal  lub Python  .

T

Ta ikona rozpoczyna fragmenty, które z jednej strony stanowią rozszerzenie głównego toku rozważań, a z drugiej — stanowią materiał trudniejszy do przyswojenia. Z obu względów, przy pierwszej lekturze książki można te fragmenty pominąć bez uszczerbku dla zrozumienia myśli przewodniej. Zachęcamy jednak do zapoznania się z nimi po przeczytaniu całego rozdziału — mogą bowiem ułatwić rozwiązanie zadań oraz problemów zamieszczonych na końcu rozdziałów i na końcu książki.



Tym znakiem poprzedzamy informacje o rozszerzeniach i uogólnieniach zagadnień, w których najczęściej odsyłamy do innych opracowań zawierających szczegółowe rozważania na ich temat.

[Harel] Nazwa w takich nawiasach oznacza pozycję w wykazie innych opracowań, skomentowanych krótko w rozdziale 14.

Każdy wyróżniony ikoną fragment tekstu oraz fragment stanowiący zamknięty blok, np. opis algorytmu czy treść ćwiczenia lub zadania w tekście, jest zakończony znakiem ■.

Zapamiętaj, że:

- ▶ na końcu rozdziału wyszczególniono **najważniejsze pojęcia, wiadomości i umiejętności**, które mogłeś zdobyć, czytając dany rozdział.
- ▶ w tym rozdziale objaśniono **wyróżnienia w tekście** oraz znaki **specjalne i ikony**, które mają zwrócić Twoją uwagę na ważniejsze fragmenty i ułatwić odszukiwanie potrzebnych informacji.

Rozdział 1. Algorytmy i sposoby ich przedstawiania

Tu dowiesz się:

- ▶ co to jest **algorytm**, chociaż nie znajdziesz ani tutaj, ani w dalszych rozdziałach tej książki, precyzyjnej definicji algorytmu;
- ▶ że książka ta nie jest tylko zbiorem algorytmów, ale przede wszystkim **prezentacją algorytmów w procesie ich powstawania** na drodze do otrzymania rozwiązania postawionego problemu;
- ▶ że algorytmy tworzą, gdy jeszcze nie było komputerów, jednak dopiero w dobie komputerów nastąpił rozwój **algorytmiki** — dziedziny zajmującej się konstruowaniem i własnościami algorytmów;
- ▶ o najważniejszych **sposobach przedstawiania algorytmów** i ich przeznaczeniu oraz własnościach.

1.1. Algorytm w procesie powstawania

Słowo **algorytm** pochodzi od brzmienia fragmentu nazwiska: Muhammad ibn Musa al-Chorezmi, arabskiego matematyka i astronoma, żyjącego na przełomie VIII i IX wieku. Jest on uznawany za prekursora metod obliczeniowych w matematyce. Od fragmentu tytułu jego dzieła pochodzi inne słowo, algebra. Upowszechnił on również stosowanie systemu dziesiętnego i posługiwanie się zerem.

Algorytm jest przepisem opisującym krok po kroku rozwiązanie problemu lub osiągnięcie jakiegoś celu. Z problemami stykamy się na każdym kroku i aby umieć je rozwiązywać, warto poznać odpowiednie algorytmy. Można wybrać jedną z dróg: spróbować rozwiązać problem samodzielnie lub skorzystać z gotowego rozwiązania. W tym drugim przypadku, gdy wybiera się gotowe rozwiązanie, dobrze jest je zrozumieć. Najczęściej znajdujemy się jednak w sytuacji, gdy dla rozważanego przez nas problemu nie ma w pełni gotowego rozwiązania, a jeśli jest — to mamy kłopoty z jego zrozumieniem. Okazuje się, że najlepszą drogą do tego, by móc rozwiązywać problemy i rozumieć ich rozwiązania, jest poznanie sposobów prowadzących do ich otrzymywania.

W tej książce przyjęliśmy właśnie takie podejście — objaśniamy rozwiązania wybranych problemów, czyli algorytmy, w trakcie ich wyprowadzania. Kładziemy więc nacisk na proces tworzenia raczej niż na sam efekt tego procesu, czyli na finalny opis metody rozwiązywania. Algorytmów nie podajemy od razu w ostatecznej postaci, ale wyprowadzamy je. Ostateczna postać byłaby

zapewne wygodna dla kogoś, kto — jak programista — chce przetłumaczyć algorytm na postać zrozumiałą dla komputera. Nam jednak zależy bardziej na przekazaniu sposobów budowania algorytmów i technik wykonywania obliczeń niż na prezentowaniu tylko samych gotowych algorytmów.

Algorytmika jest nazwą dziedziny zajmującej się algorytmami i ich własnościami. Po raz pierwszy tego terminu użył Dawid Harel w tytule swojej książki *Rzecz o istocie informatyki — algorytmika* [Harel]. Powstała ona na kanwie pogadanek o algorytmach, prowadzonych przez autora w izraelskim radiu.

Ta książka nie jest zbiorem problemów i algorytmów ich rozwiązywania, do którego można sięgnąć po gotowy przepis. Przedstawione problemy i algorytmy służą do zilustrowania ogólnych technik rozwiązywania problemów. Użyjmy pewnej analogii. Niektórzy uznają przepis kulinarny za przykład algorytmu, chociaż nie ma on wszystkich cech dobrego algorytmu (zobacz problem 13.1, a zwłaszcza rozdział 1. w książce [Piramidy]). Książka kucharska jest na ogół zbiorem pojedynczych przepisów kulinarnych, które mogą być wykorzystywane jedynie do sporządzania potraw w niej opisanych. Niniejsza książka ma być przede wszystkim źródłem metod i wskazówek do rozwiązywania problemów i konstruowania dla nich odpowiednich algorytmów. Chociaż zawiera rozwiązania wielu konkretnych problemów, główny nacisk kładziemy w niej na ogólne zasady i metody. Przyjęte podejście ma więc na celu kształtowanie i rozwijanie umiejętności rozwiązywania problemów, w tym również takich, które nie są omówione w tej książce, a mogą pojawić się przy różnych okazjach.

1.2. Algorytmy na przestrzeni wieków

Budowanie algorytmów jest tak stare, jak dążenie człowieka do rozwiązywania problemów lub zdobywania wyznaczanych celów. Człowiek, jak sięgnąć pamięcią, zawsze starał się ułatwiać sobie życie oraz ulepszać metody realizacji swoich zamierzeń, bez względu na to, czy było to wzniesienie ognia, budowanie piramid, wysyłanie wiadomości, sterowanie maszynami, porządkowanie obiektów lub wielkości, wychodzenie z labiryntu, doręczanie listów czy choćby nawet pakowanie plecaka.

Ćwiczenie 1.1. Przyjmij, że Egipcjanie, wykorzystując jedynie siłę swoich mięśni i mięśni zwierząt pociągowych, potrafili przesunąć blok kamienia po równi o niewielkiej pochyłości. Posługiwali się przy tym co najwyżej drewnianymi płozami lub rolnkami. Korzystając z tego „działania podstawowego”, zaproponuj „algorytm” zbudowania piramidy Cheopsa! Zajrzyj do rozdziału 2. w książce [Piramidy], gdzie zastanawiamy się głębiej, jak Egipcjanie budowali piramidy. ■



Ćwiczenie 1.2. Wymień przynajmniej trzy sposoby wysyłania wiadomości, które pojawiły się na przestrzeni dziejów, znacznie usprawniając komunikację między ludźmi, i do dzisiaj nie utraciły swojego znaczenia. ■



Najstarszy algorytm, który jest opisany w tej książce, algorytm Euklidesa, ma ponad 2000 lat. Wiele innych algorytmów i metod ich tworzenia opracowano przed pojawieniem się pierwszych komputerów. W dużej części były one dziełem matematyków i inżynierów, którzy opracowywali metody rozwiązywania problemów rachunkowych. Niektóre z tych metod są opisane dalej. Sporo ze współczesnych algorytmów korzysta ze zdobyczy przeszłości. Co więcej, okazało się, że często pomysły naszych przodków charakteryzowała olbrzymia intuicja i podane przez nich algorytmy są z powodzeniem stosowane również w

dobie komputerów.

Za prekursora algorytmów komputerowych jest uznawana powszechnie Ada Augusta (1815–1852), hrabina Lovelace, córka Byrona. Uważa się ją również za pierwszą programistkę komputerów — to jej imieniem nazwano jeden z nowoczesnych języków programowania wysokiego poziomu, język Ada. Zachwycona konstrukcją analitycznej maszyny Ch. Babbage’a uważała, że będzie ona „tkać wzory algebraiczne, jak krosna Jacquarda tkają liście i kwiaty”. Przełomowe znaczenie tej maszyny upatrywała „w możliwości wielokrotnego wykonywania przez nią danego ciągu instrukcji, z liczbą powtórzeń z góry zadaną lub zależną od wyników obliczeń”. Dzisiaj tak określa się podstawowe cechy algorytmów komputerowych.

Większość problemów wymaga jednak obecnie nowego spojrzenia na sposoby ich rozwiązywania. Należy uwzględniać charakter obliczeń komputerowych, w których wszystkie polecenia (działania) występujące w algorytmie muszą być możliwe do przełożenia na operacje wykonywane przez komputer, a ponadto komputery liczą na ogół niedokładnie. Pojawiają się nowe techniki algorytmiczne, których powstanie jest związane z chęcią opracowania szybszych metod rozwiązywania problemów. Powinniśmy zatem zwracać baczną uwagę na wykonalność tworzonych algorytmów oraz na ich efektywność.

Opisany w ćwiczeniu 1.3 problem znalezienia najkrótszej drogi zamkniętej przechodzącej przez każdy punkt dokładnie jeden raz nazywa się **problemem komiwojażera (TSP)**. Podane rozwiązanie jest bardzo pracochłonne — niestety, dotychczas nie wymyślono algorytmu, który znajdowałby rozwiązanie tego problemu zdecydowanie szybciej. W problemie 13.35 sugerujemy, w jaki sposób można szybko znajdować przybliżone rozwiązania problemu TSP.

Choć mogłoby się wydawać, że szybkość działania komputerów jest zawrotna, nie gwarantuje ona jednak, że każde zadanie dane komputerowi do rozwiązania będzie mogło być przez niego wykonane w „rozsądnym” czasie — niekiedy nasze życie może okazać się za krótkie, abyśmy mogli doczekać się końcowego wyniku działania programu.

Ćwiczenie 1.3. Aby się przekonać, że w powyższym stwierdzeniu nie ma przesady, wykonajmy wspólnie następujący eksperyment. Najpierw krótka historia. Zaplanowano zorganizowanie trzech spotkań premiera z wojewodami w ich siedzibach. W pierwszym — mają wziąć udział wojewodowie z tych województw, które są w obecnym podziale administracyjnym — jest ich 16, w drugim — z tych, które były w planowanym podziale — miało ich być 25, a w trzecim — z wszystkich województw, które były w latach 1975-1998 — było ich 49. Premier ma wyruszyć z Warszawy, odwiedzić wszystkie województwa, każde dokładnie jeden raz, i wrócić do stolicy. Nasze zadanie polega na wskazaniu premierowi najkrótszej drogi. Na rysunku 1.1 przedstawiono

optymalną pod względem długości trasę objazdu wszystkich województw z nowego podziału administracyjnego.



Rysunek 1.1. Najkrótsza trasa premiera przebiegająca przez wszystkie województwa w obecnym podziale administracyjnym

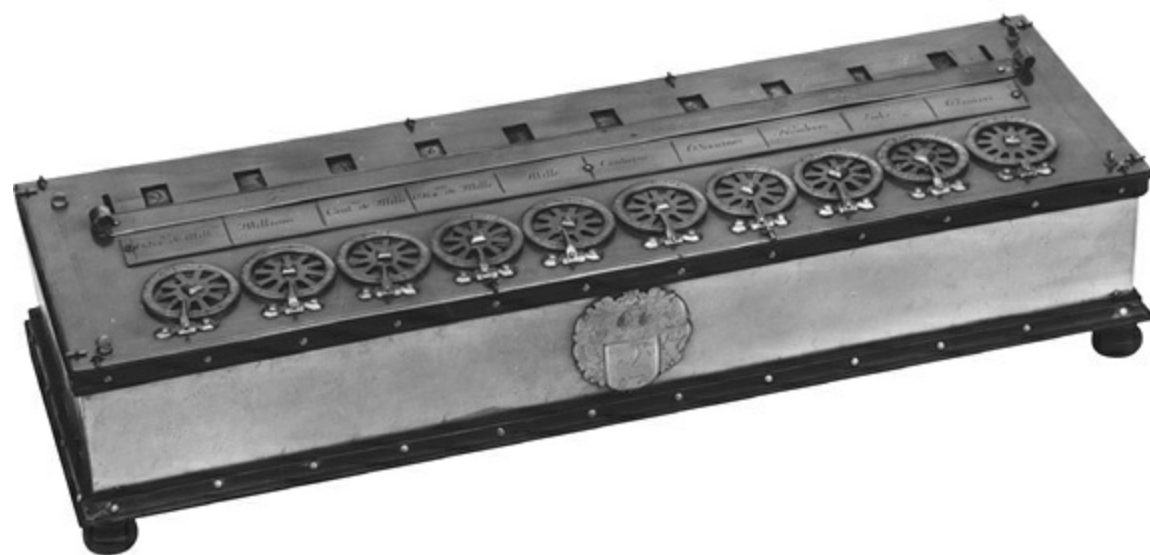
Przypuśćmy, że nie dysponujemy żadnym innym algorytmem i jedyna metoda, jaka przychodzi nam do głowy, polega na przejrzaniu wszystkich możliwych ustawień województw na drodze premiera. Jeśli n jest liczbą województw, to istnieje $(n - 1)! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 1)$ możliwych ich ustawień, gdyż poza stolicą, od której rozpoczyna się wizyta, pozostałe województwa mogą wystąpić w dowolnej kolejności (zobacz problem 13.2). Przypuśćmy teraz, że dysponujemy superkomputerem, który w ciągu sekundy przegląda 10^{15} ustawień województw i wybiera spośród nich to, dla którego trasa premiera jest najkrótsza. Jak długo będzie trwało znalezienie optymalnej drogi objazdu wszystkich województw w tych trzech przypadkach? W tabeli 1.1 przedstawiamy wyniki. By uwierzyć, że autor nie pomylił się w rachunkach, proponujemy sprawdzić na kalkulatorze podane w niej liczby. ■

Tabela 1.1. Czasy znalezienia przez komputer, wykonujący 1 biliard operacji na sekundę, najkrótszej trasy podróży premiera (zobacz ćwiczenie 1.3)

Liczba województw	Czas znalezienia najkrótszej trasy
-------------------	------------------------------------

16	0.0002 sek.
25	20 lat
49	$4 \cdot 10^{38}$ lat

Równocześnie z tworzeniem algorytmów człowiek starał się budować urządzenia, które pomogłyby mu wykonywać obliczenia. Pierwsze takie urządzenia były liczydłami — konstruowali je Chińczycy, Rzymianie, Arabowie. W XVII wieku zaczęto budować urządzenia, które nazwalibyśmy dzisiaj kalkulatorami — ich twórcami byli m.in. Blaise Pascal (1623 - 1662) i Gottfried W. Leibniz (1646 - 1716). Od początku XIX wieku rozpoczyna się era maszyn budowanych na zasadach, które odnajdujemy w obecnych komputerach — pierwszym wielkim konstruktorem takich maszyn był Charles Babbage (1791 - 1871). Jednak twórcą komputera w dzisiejszym sensie jest Konrad Zuze (1910 - 1995) — pierwszą swoją maszynę zbudował on jeszcze przed II wojną światową.



Pascalina — arytmetr zbudowany przez Pascala



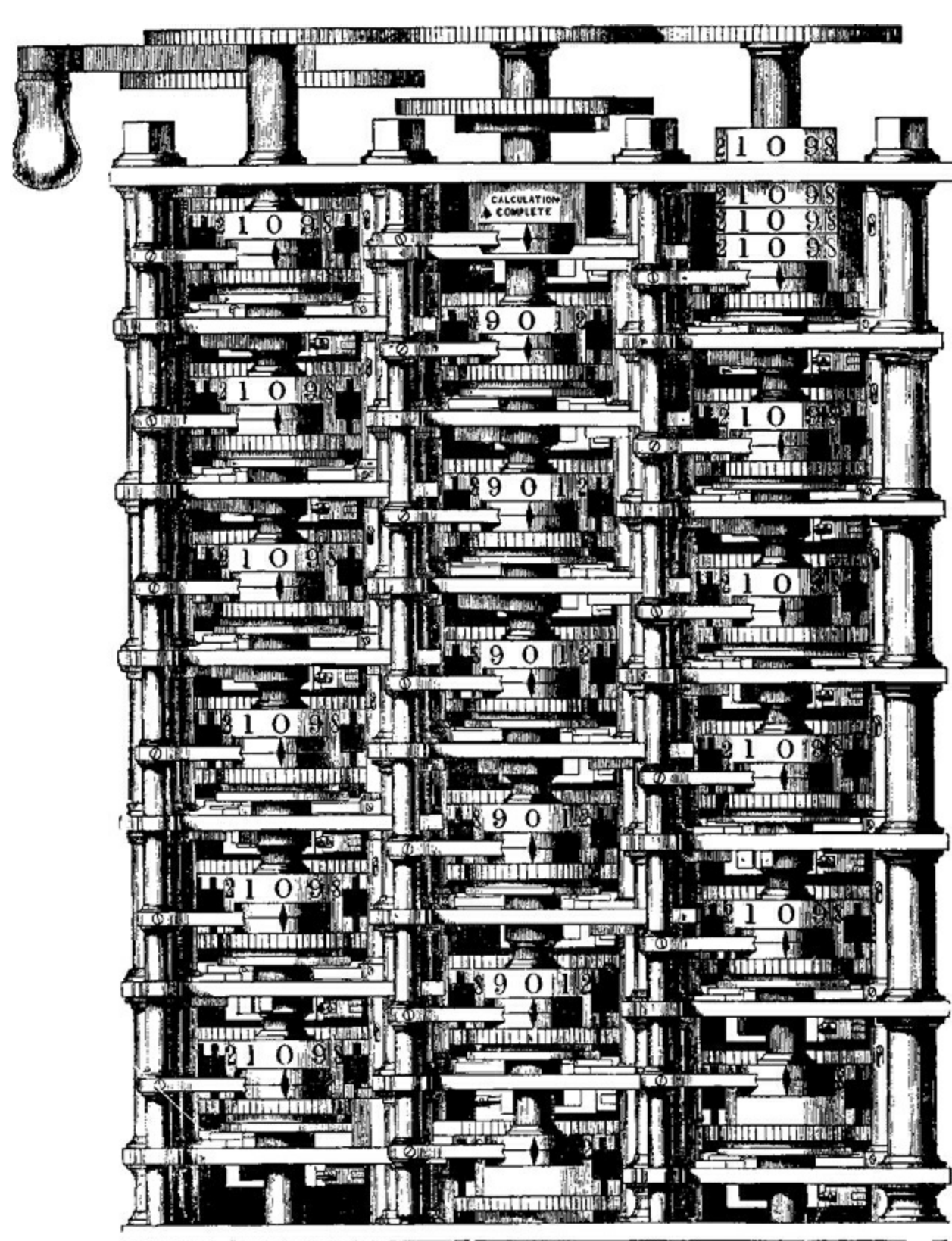
Zachęcamy do zapoznania się z wybranymi elementami historii komputerów i informatyki, w której przeplata się historia konstruowania coraz doskonalszych maszyn liczących z historią tworzenia coraz efektywniejszych algorytmów dla tych maszyn. Znajomość historii informatyki pomaga zrozumieć teraźniejszość i tendencje w rozwoju tej dziedziny. Najważniejsze jej wyjątki można znaleźć w podręczniku [EI-I]. Polecamy również bardzo bogato ilustrowaną historię informatyki w postaci plansz historycznych [Plansze Historia]. ■

Najlepszym sposobem przyspieszania pracy komputerów jest obarczanie ich

mniejszą liczbą działań.

Ralph Gomory (IBM)

Analiza tempa zwiększania szybkości działania komputerów, które jest jednak dość powolne, i konieczność uwzględnienia rosnących potrzeb użytkowników komputerów skłoniły Ralpha Gomory'ego — byłego szefa naukowego koncernu IBM — do wypowiedzenia słów zamieszczonych obok na marginesie. Oznaczają one, że za pomocą komputerów można rozwiązywać problemy coraz szybciej — przede wszystkim dzięki stosowaniu bardziej efektywnych algorytmów, czyli wykonujących mniejszą liczbę operacji.



Fragment maszyny różnicowej Babbage'a

1.3. Reprezentacje problemów i algorytmów

Reprezentacje algorytmów są nie tylko ich opisami, ale często wykorzystuje się je przy konstruowaniu algorytmów oraz badaniu własności wykonywanych przez nie obliczeń. Żaden ze sposobów zapisywania algorytmów nie jest

faworyzowany w tej książce. Dany algorytm jest opisany w najbardziej odpowiedni dla niego sposób, który zależy od rodzaju wykonywanych w nim operacji oraz możliwości wybranej reprezentacji. Na przykład nie każdy algorytm można zapisać w postaci drzewa obliczeń lub drzewa wyrażeń. Jeśli z kolei planuje się wykonywanie eksperymentów komputerowych z algorytmem, to właściwym opisem może być program komputerowy. Poszczególne rodzaje opisów algorytmów wprowadzamy tylko w takim zakresie, w jakim będą nam potrzebne w konkretnym przypadku. W rozdziale 14. wymieniamy opracowania, w których dokładnie przedstawiono poszczególne sposoby reprezentowania algorytmów. Ponieważ głównym celem tej książki jest przedstawienie algorytmów w procesie ich konstruowania, mniejszą wagę przywiązujemy do ich opisów w postaci programów komputerowych. Chcemy w ten sposób osiągnąć jeszcze jeden cel — by o algorytmach myśleć jak o sposobach rozwiązywania problemów, niezależnych od wybranej reprezentacji oraz systemu programowania, w którym algorytm ma być wykonany za pomocą komputera. Algorytmiczne myślenie można bowiem kształtować niezależnie od programowania komputerów, chociaż każdy program komputerowy jest zapisem jakiegoś algorytmu.

1.3.1. Roboczy przykład

Posłużymy się następującym przykładem zadania rachunkowego, dla którego utworzymy algorytm i przedstawimy różne jego reprezentacje.

Oblicz wartość funkcji:

$$f(x) = \frac{x}{|x|} \quad (1.1)$$

dla dowolnej liczby rzeczywistej x , gdzie $|x|$ oznacza funkcję — wartość bezwzględną, zwaną również **modułem**. Naszym zadaniem jest podanie algorytmu, czyli sposobu obliczania wartości tej funkcji. Już na tym prostym przykładzie widać, że przed przystąpieniem do opisanego sposobu rozwiązania postawionego zadania musimy najpierw zadbać o jego dokładne sformułowanie. W informatyce używa się pojęcia **specyfikacja** na oznaczenie dokładnego opisu zadania, które ma być wykonane (lub problemu, który ma być rozwiązany). Na ogół specyfikacja ma postać wyszczególnienia, w którym są wymienione **dane** dla zadania i warunki, jakie muszą one spełniać, oraz **wyniki** i również warunki, jakie muszą one spełniać, a ściślej — jaki jest związek wyników z danymi. Aby podać specyfikację zadania polegającego na obliczaniu wartości funkcji (1.1), musimy uzupełnić jej opis o dziedzinę, czyli dla jakich wartości x funkcja ta ma określone wartości. Z postaci wzoru wynika, że x nie może być równe zero. Ustalmy jednak dodatkowo, że wartością funkcji f w zerze jest 0. Zatem nasza

funkcja jest określona dla wszystkich liczb rzeczywistych x . Możemy już teraz podać specyfikację rozwiązywanego zadania:

Obliczanie wartości przykładowej funkcji

Dane: Dowolna liczba rzeczywista x .

Wynik: Wartość funkcji $f(x)$ podanej wzorem (1.1), jeśli x jest różne od zera i 0 — w przeciwnym wypadku. ■

W tej specyfikacji wyszczególniliśmy, że daną w problemie jest jedna liczba rzeczywista, a wartością — również liczba rzeczywista. Dodatkowo z określenia wyniku wiemy, że jeśli dana jest różna od zera, to wynik jest określony wzorem (1.1), a jeśli $x = 0$, to wynikiem jest również 0.

W następnych punktach tego rozdziału wprowadzamy różne reprezentacje algorytmu rozwiązywania zadania o tej specyfikacji.

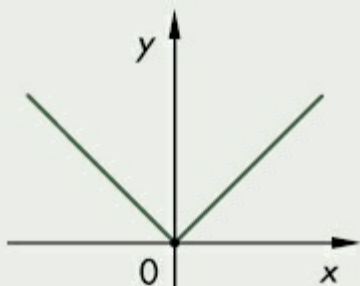
1.3.2. Słowny opis algorytmu

Słowny opis algorytmu jest na ogół jego pierwszym, mało ścisłym opisem. Rozpoczyna się często dyskusją, w jaki sposób można rozwiązać postawione zadanie, i służy wyrobieniu pewnej intuicji oraz ukierunkowaniu rozważań na właściwe sposoby i techniki przydatne w rozwiązaniu.

Dyskusję o sposobie obliczania wartości funkcji danej wzorem (1.1) rozpoczęliśmy już w poprzednim punkcie. Teraz zastanówmy się, czy rzeczywiście musimy wykonywać operacje określone tym wzorem. Przypomnijmy sobie najpierw definicję funkcji moduł — wartość bezwzględna $|x|$. Wiadomo z matematyki, że:

$$|x| = \begin{cases} -x, & \text{dla } x < 0 \\ x, & \text{dla } x \geq 0 \end{cases}$$

Wykres funkcji $y = |x|$



Wynika stąd, że funkcja $f(x)$ dana wzorem (1.1) ma wartość 1 dla liczb dodatnich i -1 dla liczb ujemnych. Na tej podstawie możemy zmienić specyfikację naszego

zadania, która teraz przyjmuje następującą postać:

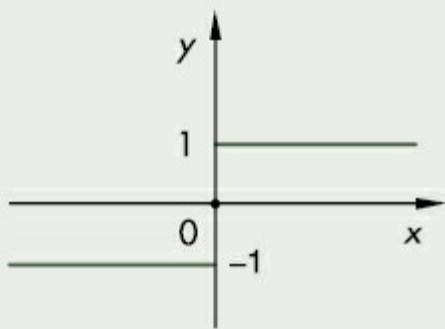
Obliczanie wartości przykładowej funkcji

Dane: Dowolna liczba rzeczywista x .

Wynik: Wartość funkcji $f(x)$ określonej następującym wzorem:

$$f(x) = \begin{cases} -1, & \text{dla } x < 0 \\ 0, & \text{dla } x = 0 \\ 1, & \text{dla } x > 0 \end{cases} \quad (1.2) \blacksquare$$

Wykres funkcji (1.2)



Na tym można zakończyć słowny opis metody rozwiązania postawionego zadania i uściślenia jego specyfikacji.

1.3.3. Opis algorytmu w postaci listy kroków

Lista kroków jest jednym z najczęściej stosowanych, dokładnych sposobów opisywania obliczeń oraz ich kolejności. Poszczególne kroki zawierają opis operacji, które mają być wykonane przez algorytm. Mogą w nich również wystąpić polecenia związane ze zmianą kolejności wykonywania kroków lub polecenia zakończenia algorytmu. Przyjmuje się w tym opisie, że kolejność wykonywania poszczególnych kroków jest zgodna z kolejnością ich wypisania — z wyjątkiem sytuacji, gdy jednym z poleceń w kroku jest przejście do wykonania kroku o podanym numerze.

Dla ścisłości opisu, między nazwą algorytmu a listą jego kroków podajemy specyfikację problemu rozwiązywanego przez ten algorytm. Dodatkowo w nawiasach $\{ \dots \}$ możemy umieścić uwagi nie będące częścią algorytmu, a jedynie komentujące jego przebieg i pomagające zrozumieć wykonywane polecenia i ich efekty.

Dla przykładowego problemu algorytm w postaci listy kroków możemy zapisać następująco:

Algorytm obliczania wartości przykładowej funkcji (1.2)

Dane: Dowolna liczba rzeczywista x .

Wynik: Wartość funkcji $f(x)$ określonej wzorem (1.2).

Krok 0. Wczytaj wartość danej x .

Krok 1. Jeśli $x > 0$, to $f(x) = 1$. Zakończ algorytm.

Krok 2. {W tym przypadku $x \leq 0$.} Jeśli $x = 0$, to $f(x) = 0$. Zakończ algorytm.

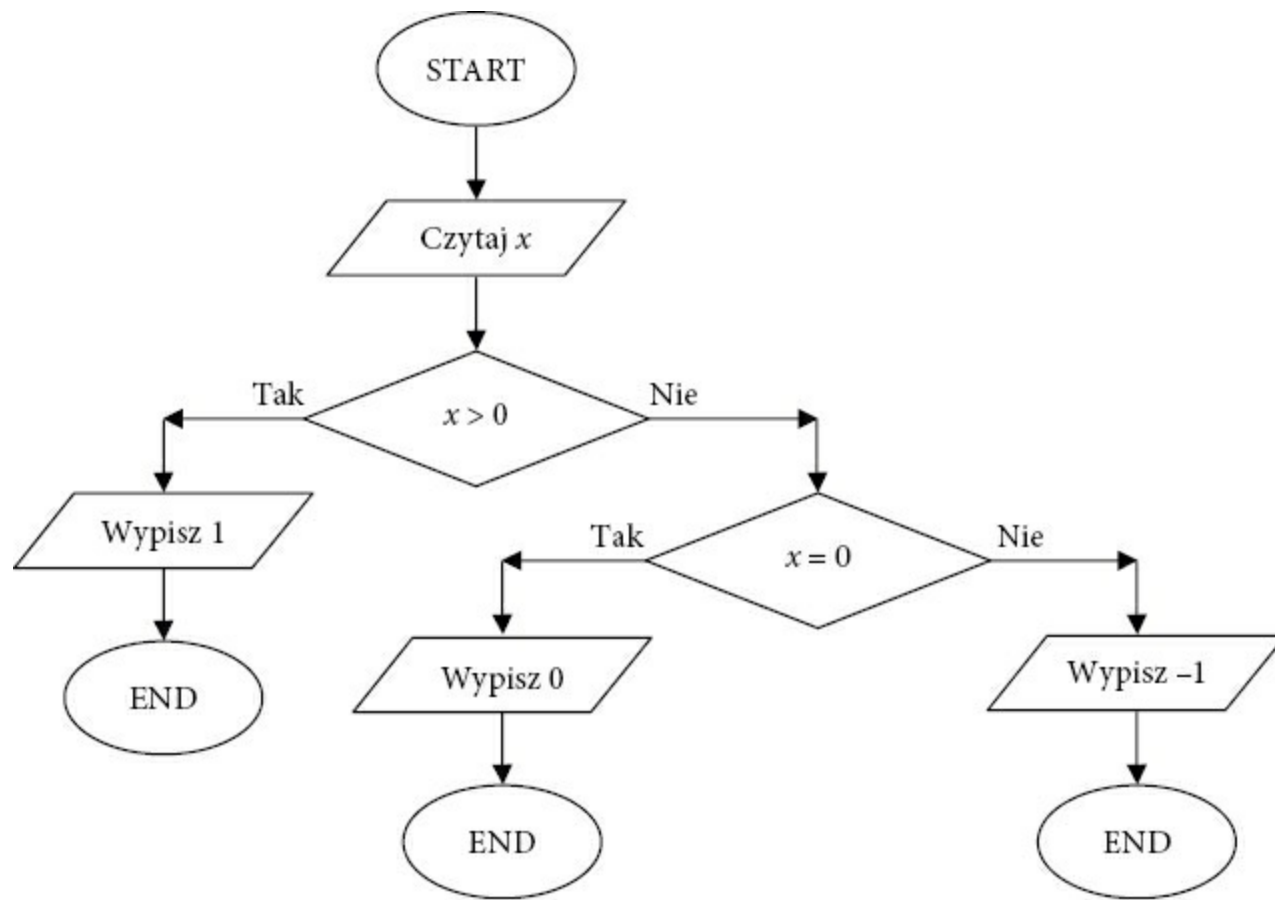
Krok 3. {W tym przypadku $x < 0$.} Mamy $f(x) = -1$. Zakończ algorytm. ■

Ćwiczenie 1.4. Sprawdź dla $x = -1, 0$ i 1 , że ten algorytm poprawnie oblicza wartość funkcji (1.2). ■

Opis algorytmu w postaci listy kroków występuje w tej książce najczęściej, gdyż najdokładniej określa wykonywane w algorytmie działania i ich kolejność. Aby nieco uprościć zapis algorytmów w tej postaci, będziemy opuszczać krok 0., polegający na wczytaniu danych do algorytmu. Będziemy jednak zakładać, że dane wyszczególnione w specyfikacji problemu są dostępne dla algorytmu i spełniają opisane w niej warunki.

1.3.4. Schemat blokowy algorytmu

Schemat blokowy jest jednym z najbardziej popularnych, graficznych sposobów przedstawiania algorytmów. Składa się z bloków oraz połączeń między nimi. W blokach są zapisywane operacje, które mają być wykonane, a połączenia wyznaczają kolejność ich wykonywania. Często stosuje się różne kształty bloków dla odróżnienia rodzajów operacji w nich zawartych (zobacz rysunek 1.2).

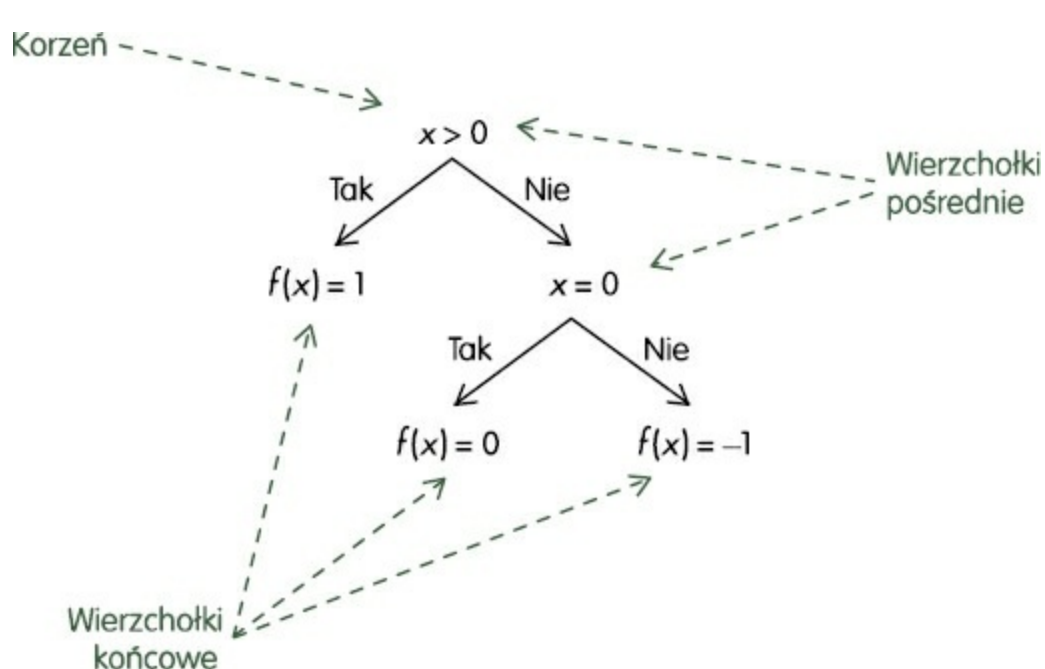


Rysunek 1.2. Schemat blokowy algorytmu obliczania wartości funkcji (1.2)

1.3.5. Drzewo algorytmu

Drzewo w opracowaniach informatycznych jest rysowane na ogół do góry nogami, z korzeniem u góry. Stosowane nazewnictwo elementów drzewa jest wzięte z dendrologii — **korzeń**, wierzchołki końcowe nazywają się **liśćmi**, a krawędzie, czyli połączenia w drzewie — **gałęziami**.

Drzewo algorytmu, nazywane również drzewem obliczeń, jest szczególnym rodzajem schematu blokowego, który przyjmuje postać drzewa. Oznacza to, że każde dwie drogi obliczeń w takim schemacie mogą mieć tylko początkowe fragmenty wspólne, ale po rozejściu już się nie spotykają. Schemat blokowy na rysunku 1.2 ma właśnie postać drzewa algorytmu. Najczęściej drzewa algorytmów przedstawia się, upraszczając nieco ich postać graficzną — na rysunku 1.3 jest pokazane drzewo odpowiadające schematowi z rysunku 1.2. W drzewie algorytmu można wyróżnić: **korzeń** — wierzchołek, w którym rozpoczynają się działania algorytmu, **wierzchołki pośrednie**, w których są umieszczone operacje wykonywane w algorytmie, oraz **wierzchołki końcowe (liście)**, które odpowiadają różnym wynikom zakończenia obliczeń w algorytmie. Ten sposób przedstawiania algorytmów jest bardzo dogodny do ich analizy, a zwłaszcza do wyznaczania liczby wykonywanych operacji.

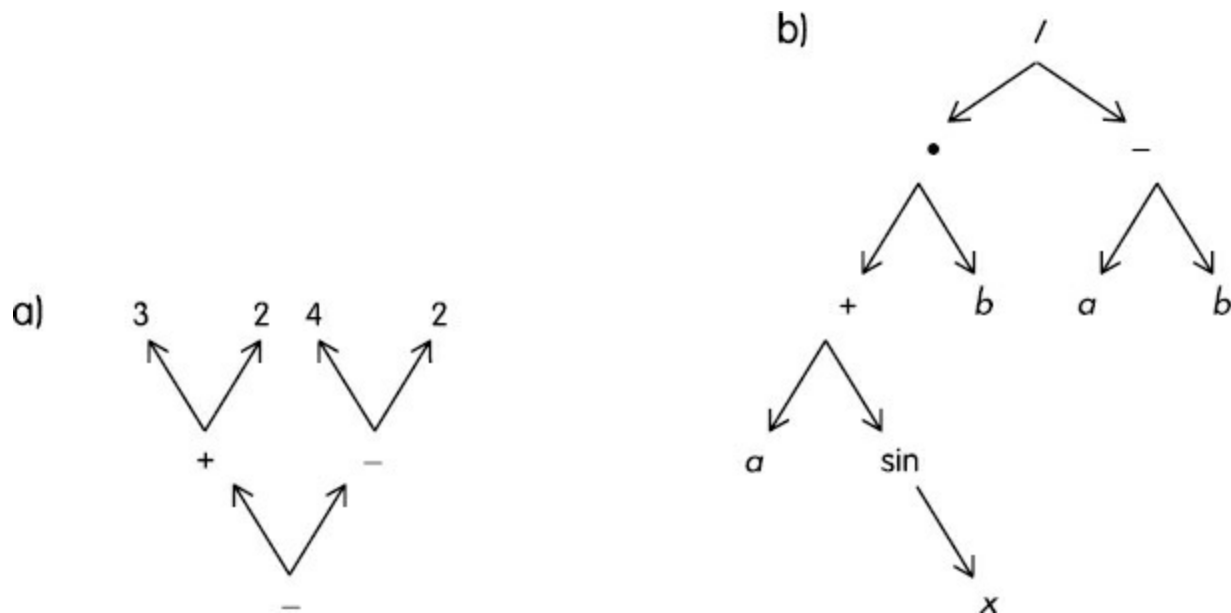


Rysunek 1.3. Drzewo algorytmu obliczania wartości funkcji (1.2), odpowiadające schematowi blokowemu z rysunku 1.2

1.3.6. Drzewo wyrażenia

Drzewo wyrażenia ma ograniczone zastosowania jako reprezentacja algorytmu — jest wykorzystywane przede wszystkim do obliczania wartości wyrażeń. Z tą reprezentacją wyrażeń spotykamy się w szkole bardzo wcześnie, podczas nauki obliczania wartości wyrażenia arytmetycznego zawierającego nawiasy.

Na rysunku 1.4a jest przedstawiony przykład takiego drzewa — to pierwsze drzewo jest rysowane tradycyjnie, z korzeniem u dołu. Jest w nim zapisane wyrażenie $(3 + 2) - (4 - 2)$. W drzewie można zapisać dowolne wyrażenie — zobacz na rysunku 1.4b drzewo innego wyrażenia, narysowane już w sposób informatyczny, korzeniem do góry. W takich drzewach argumenty wyrażeń znajdują się w liściach, a działania — w wierzchołkach pośrednich. Obliczanie wartości wyrażenia zapisanego w drzewie niejako spływa od liści, gałęziami ku korzeniowi (w dół lub do góry, w zależności od sposobu rysowania drzewa).



Rysunek 1.4. Drzewa wyrażeń: a) $(3 + 2) - (4 - 2)$; b) $(a + \sin x) \cdot b / (a - b)$

1.3.7. Program zapisany w języku Pascal lub Python

Program napisany w języku programowania jest najbardziej ścisłym i zrozumiałym dla komputera opisem algorytmu. Tekst programu może również służyć do komunikowania rozwiązania człowiekowi. Wszystkie algorytmy przedstawione w tej książce zostały opisane w postaci procedur (lub programów) w językach Pascal i Python, a ich teksty znajdują się w towarzyszących jej zasobach elektronicznych na stronie <http://edukacja.helion.pl/algorytmika>. W książce zamieszczamy tylko kilka takich opisów, gdyż w niewielkim stopniu posługujemy się programami w konkretnym języku programowania jako reprezentacjami algorytmów.

Przeznaczenie opisów algorytmów w języku programowania jest dwojakie — mogą być one czytane jako ścisłe opisy algorytmów lub wykorzystywane w obliczeniach komputerowych, gdy nauka o algorytmach obejmuje również programowanie. Zachęcamy również tych, którzy nie znają ani języka Pascal, ani języka Python lub nie mają dostatecznej wprawy w posługiwaniu się nimi, by próbowali odczytać opis algorytmu w programach w tych językach. Poniżej przedstawiamy zapis w językach Pascal i Python obliczania wartość funkcji określonej wzorem (1.2).



```
program Funk1_2;
    var x: real;

begin
```



```
read(x);  
if x > 0 then write(1)  
else  
    if x = 0 then write(0)  
    else write(-1)  
end. {Funk1_2}
```



```
def Funk1_2(x):  
    print("Wartość funkcji (1.2)")  
    print("dla x =", x, end="")  
    print("równa się =", end="")  
    if x > 0:  
        print(1)  
    elif x == 0:  
        print(0)  
    else:  
        print(-1)
```

1.4. Ćwiczenia, zadania, problemy

W książce tej zachęcamy Czytelnika do udziału w dyskusji o rozwiązaniach problemów i o algorytmach poprzez zadawanie mu pytań i stawianie zadań do rozwiązania. Chcemy tym samym osiągnąć nasz główny cel: przekonać, że najlepszym sposobem poznania i zrozumienia algorytmu jest przebycie lub przynajmniej prześledzenie drogi jego powstania. Skala trudności zadań stawianych Czytelnikowi oraz spodziewany zakres samodzielności przy ich rozwiązywaniu są różne, postanowiliśmy więc zaznaczać to w odpowiedni sposób.

Ćwiczenie jest poleceniem wykonania prostych czynności i na ogół pojawia się w trakcie konstruowania algorytmu — angażując Czytelnika do współtworzenia rozwiązania. Może również polegać na sprawdzeniu, czy otrzymane rozwiązanie jest zrozumiałe, poprzez wykonanie algorytmu dla konkretnych danych lub utworzenie prostej jego modyfikacji. Ćwiczenia znajdują się w tekście poszczególnych punktów.

Rozwiązanie zadania polega na skorzystaniu z poznanych już rozwiązań lub algorytmów albo wykonaniu ich modyfikacji. Na ogół, w treści zadania jest podane, z czego należy skorzystać lub jaka powinna być postać rozwiązania.

Zadania pojawiają się w tekście poszczególnych punktów lub na końcu rozdziałów.

Każdy problem jest dla nas wyzwaniem, ale wyposażeni w odpowiednie metody nie powinniśmy traktować żadnego z nich jako trudności nie do pokonania. Pięknie pisze o tym George Pólya [Pólya]: *...rozwiązanie każdego problemu ma pewne cechy odkrycia. Wasz problem może być skromny; jeśli jednak zaciekawia on Was i pobudzi do czynu Wasze zdolności twórcze i jeśli rozwiążecie go własnymi siłami, to możecie doznać emocji towarzyszącej napięciu umysłu i triumfowi dokonanego odkrycia.*

Problem jest w naszej skali najtrudniejszym do wykonania poleceniem. W większości problemów to osoba rozwiązująca go musi zdecydować, z czego i jak ma skorzystać. Czasem może w tym pomóc nagłe olśnienie lub nietypowy pomysł. Problemy zamieszczone na końcu rozdziału na ogół można rozwiązać, posługując się wiadomościami zawartymi w tym rozdziale. W rozdziale 13. zostały natomiast zebrane problemy, do których rozwiązania są potrzebne informacje i wiadomości podawane w wielu fragmentach tej książki, oraz problemy, które mogą stanowić niemałe wyzwanie dla Czytelnika.

Jeszcze jedna uwaga terminologiczna dotycząca znaczenia określenia „problem” — używamy tego terminu również w znaczeniu popularnym, odnoszącym się do zadania postawionego do rozwiązania, a więc występuje on tutaj w znaczeniu przyjętym w dziedzinie rozwiązywania problemów.

Mogłeś dowiedzieć się, że:

- **algorytm** jest dokładnym przepisem rozwiązania problemu lub osiągnięcia zamierzonego celu;
- **algorytmika** jest dziedziną wiedzy, zajmującą się budowaniem algorytmów oraz wszechstronnym poznawaniem i badaniem ich własności;
- znaczenie algorytmów w dobie komputerów jest szczególne, ponieważ każdy **program komputerowy działa według** jakiegoś **algorytmu**;
- sposób opisu algorytmu należy dobrać w zależności od rodzaju rozwiązywanego problemu oraz jego przeznaczenia;
- graficzny opis algorytmu można wykorzystać do symulacji jego działania lub badania własności, bez konieczności pisania programu komputerowego.

Rozdział 2. Algorytmy liniowe

Tu dowiesz się:

- ▶ na czym polegają **obliczenia** (algorytmy) **liniowe**;
- ▶ czy rzeczywiście trzeba zajmować się tak prostymi obliczeniami.

Algorytm liniowy ma postać ciągu (listy) kroków, które bezwarunkowo powinny być wykonane zgodnie z kolejnością, w jakiej występują. Taki algorytm nie może więc zawierać sprawdzania warunków, od których spełnienia zależy kolejność wykonywania kroków. Zatem przedstawiony w poprzednim rozdziale algorytm obliczania wartości przykładowej funkcji nie jest w pełni liniowy. Schemat blokowy algorytmu liniowego jest po prostu jednym ciągiem bloków.

Czy algorytmy liniowe są na tyle ciekawe, by się nimi zajmować? Czy mogą pomóc rozwiązywać jakieś poważniejsze zadania lub problemy, zwłaszcza gdy dodatkowo dysponujemy komputerem? Już przecież Ada, hrabina Lovelace, zachwycając się projektem maszyny analitycznej Babbage'a, miała powiedzieć, że znaczenia komputerów (a więc i algorytmów) należy upatrywać przede wszystkim w ich możliwościach wielokrotnego wykonywania ciągu instrukcji.

Odpowiedzi na te pytania i wątpliwości zależą oczywiście od stopnia złożoności obliczeń do wykonania. W tej książce zamieszczamy jedynie proste przykłady algorytmów w pełni liniowych. Weźmy jednak pod uwagę obliczenia wykonywane w arkuszach kalkulacyjnych. Większość z nich to rachunki przebiegające według algorytmów liniowych. Co więcej, chcąc wielokrotnie wykonać ciąg tych samych instrukcji w arkuszu, należy je rozpisać na ustaloną z góry liczbę wierszy lub kolumn, czyli przedstawić w postaci algorytmu liniowego. Dodajmy tutaj jeszcze, że każdy algorytm przedstawiony w dalszych rozdziałach zawiera fragmenty mające strukturę obliczeń liniowych.

Algorytmy liniowe są zapisem obliczeń, które mają postać ciągu operacji rachunkowych (matematycznych) wykonywanych bez sprawdzania jakichkolwiek warunków. Podamy teraz algorytm liniowy dla bardzo prostego zadania, które dodatkowo jeszcze uprościmy.

Wielomiany występują w wielu szkolnych zadaniach z matematyki oraz fizyki i często obliczamy ich wartości dla konkretnych argumentów. Na ogół są to wielomiany drugiego lub trzeciego stopnia, a więc funkcje tak proste, że bez zastanowienia wykonujemy zapisane w nich działania. Ale czy rzeczywiście postępujemy najprościej? Rozważmy wielomian drugiego stopnia:

$$w(x) = ax^2 + bx + c.$$

Aby obliczyć jego wartość, zwykle podnosimy najpierw x do kwadratu (czyli mnożymy x przez siebie), a następnie dwa razy mnożymy przez współczynniki a i

b oraz dwa razy dodajemy. W sumie wykonujemy w ten sposób trzy razy mnożenie i dwa razy dodawanie. Zauważmy jednak, że po zgrupowaniu dwóch pierwszych wyrazów i wyłączeniu wspólnego czynnika x wielomian stopnia drugiego można przedstawić w następującej postaci:

$$w(x) = (ax + b)x + c.$$

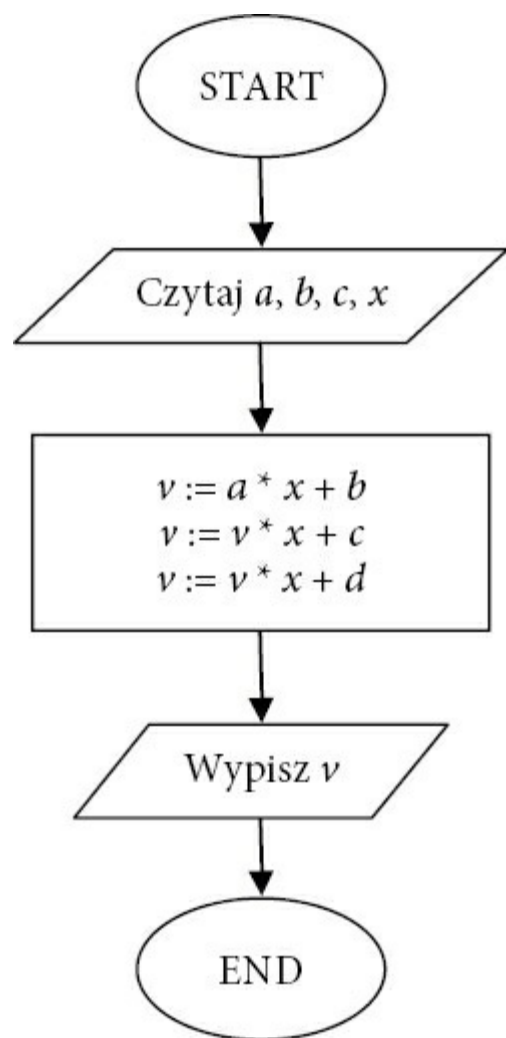
Postacie wielomianów drugiego i trzeciego stopnia, które tutaj wyprowadzamy, oraz wynikające z tych postaci sposoby obliczania wartości tych wielomianów są szczególnym przypadkiem tak zwanego **schematu Hornera** (punkt 7.2).

Jeśli skorzystamy z tej postaci, to obliczenie wartości wielomianu drugiego stopnia polega na wykonaniu już tylko dwóch działań mnożenia i nadal dwóch dodawania, czyli o jedno mnożenie mniej niż poprzednio. Zysk niewielki, ale staje się bardziej widoczny, gdy w ten sam sposób przekształcimy wielomian trzeciego stopnia, by obliczać jego wartości:

$$v(x) = ax^3 + bx^2 + cx + d = ((ax + b)x + c)x + d.$$

Obliczenie wartości wielomianu $v(x)$ z pierwszego wzoru wymaga wykonania pięciu działań mnożenia i trzech dodawania, natomiast w przypadku drugiego wzoru — wystarczą jedynie trzy mnożenia i trzy dodawania.

Na rysunku 2.1 jest przedstawiony schemat blokowy, służący do obliczania wartości wielomianu $v(x)$. W bloku wykonania obliczeń występuje **instrukcja przypisania**, czyli polecenie stosowane do obliczania wartości wyrażeń. W tej instrukcji w języku Pascal używa się **znaku przypisania** $:=$, a w języku Python — zwykłego znaku równości $=$. Działanie tej instrukcji polega na obliczeniu wartości wyrażenia stojącego po prawej stronie znaku przypisania i przypisaniu tej wartości zmiennej stojącej po lewej stronie tego znaku. Stąd wynika, że przed znakiem przypisania musi być umieszczona nazwa zmiennej. W instrukcji przypisania nazwa jednej zmiennej może więc wystąpić zarówno po lewej, jak i po prawej stronie znaku przypisania. Stąd, chociaż w matematyce nie jest prawdziwa równość $n = n + 1$, w opisach algorytmów można stosować instrukcję $n := n + 1$ (w języku Pascal) lub $n = n + 1$ (w języku Python), której wykonanie powoduje zwiększenie wartości zmiennej n o 1.



Rysunek 2.1. Schemat blokowy algorytmu liniowego, służącego do obliczania wartości wielomianu $v(x)$.

Zwróćmy uwagę na podobieństwo między kolejnymi krokami obliczeń w bloku wykonania obliczeń. Wykorzystamy to później (w punkcie 7.2) przy wyprowadzaniu ogólnej postaci schematu Hornera. ■

Ćwiczenie 2.1. Zmodyfikuj schemat blokowy z rysunku 2.1 tak, aby w bloku wykonania obliczeń znalazła się tylko jedna instrukcja przypisania. Czy można z tego schematu całkowicie usunąć blok wykonania obliczeń? Sprawdź w tym celu, jakie są możliwości bloku wyprowadzania wyniku. ■

2.1. Zadania

Zadanie 2.1. Ile razy musisz wykonać mnożenie, aby obliczyć wartość wyrażenia x^6 ? Pięć razy? Wystarczy wykonać trzy razy, skorzystaj w tym celu z przedstawienia $6 = 2 \cdot 3$. Podaj inny sposób obliczania szóstej potęgi, polegający na wykonaniu również tylko trzech działań mnożenia, który wynika z rozwinięcia liczby 6 w systemie dwójkowym. Zapisz te sposoby w postaci algorytmów.

Zadanie 2.2. Dysponujesz w banku na swoim koncie kwotą p złotych. Zaoferowano Ci umieszczenie tej kwoty na koncie oprocentowanym $x\%$ w skali rocznej i dopisywanie oprocentowania co trzy miesiące. Jakiej kwoty możesz się spodziewać na swoim koncie po trzech i sześciu miesiącach oraz po roku i po trzech latach? Opisz odpowiedni algorytm postępowania.

Wskazówka. Czy pamiętasz, że po roku na Twoim koncie będzie kwota $(1 + x)p$?

Powinieneś wiedzieć:

- ▶ jakie cechy ma **algorytm liniowy**;
- ▶ że obliczenia według algorytmów liniowych stanowią pokaźny procent wszystkich obliczeń komputerowych.

Rozdział 3. Algorytmy z rozgałęzieniami

Tu poznasz:

- ▶ **sposób zabezpieczania się przed zabronionymi działaniami matematycznymi w algorytmach;**
- ▶ **algorytm znajdowania pierwiastków równania kwadratowego, który uwzględnia, że stosujesz do tego komputer;**
- ▶ **sposób zorganizowania obliczeń rozgałęziających się na wiele przypadków**

oraz dowiesz się:

- ▶ że w niektórych przypadkach **nie można uniknąć zgrubnych obliczeń** wykonywanych na komputerze.

Znamy wiele przykładów obliczeń w matematyce lub w fizyce, w których wykonanie niektórych kroków zależy od spełnienia warunku. Posłużyliśmy się już takim przykładem w punkcie 1.3, opisując dla niego różne sposoby reprezentowania algorytmów.

Sprawdzanie warunku występuje w wielu wzorach matematycznych, w których działania mają sens i mogą być wykonane tylko na liczbach spełniających pewne warunki. Wiadomo na przykład, że nie wolno dzielić przez zero i dlatego w algorytmie, w którym występuje dzielenie, należy upewnić się, czy dzielnik nie jest równy zero. Podobnie umiemy obliczać pierwiastek kwadratowy tylko z liczb nieujemnych i dlatego przed obliczeniem wartości pierwiastka musimy sprawdzić, czy liczba podpierwiastkowa jest nieujemna.

Te zabezpieczenia, polegające na sprawdzaniu warunków, mają jeszcze większe znaczenie w obliczeniach, które są wykonywane automatycznie, np. za pomocą komputerów. Maszyny te „nie widzą”, nie odróżniają więc dobrych od złych danych i na tych drugich mogą również próbować wykonywać obliczenia, gdy im je podsunie. Ale dzielić przez zero nie potrafi nawet komputer, więc wstrzymuje działanie i „zawiesza się”. Dlatego w algorytmach, których jedną z cech ma być niezawodność, musimy umieć zabezpieczać się przed zabronionymi działaniami, by pewne operacje mogły być wykonywane tylko wtedy, gdy są spełnione odpowiednie warunki.

W tym rozdziale omawiamy wykonywanie obliczeń, które zależą od spełnienia pewnych warunków. Wykorzystujemy w tym celu proste zadania znane z matematyki szkolnej, w których należy zapewnić, że działania matematyczne mogą być wykonane. Podajemy również złożony przykład analizy

podprzypadków, w którym występuje duże nagromadzenie kroków warunkowych. Zadania matematyczne posłużą ponadto do przedstawienia przykładów obliczeń, w których może dojść do utraty dokładności obliczeń i ostatecznych wyników.

Inne algorytmy, w których są wykonywane tylko porównania, przedstawiamy w rozdziale 4., zajmując się porządkowaniem kilku liczb. W następnych rozdziałach niemal każdy algorytm zawiera kroki warunkowe; występują one m.in. w wykonywaniu powtarzających się obliczeń, bez względu na to, czy powtarzanie jest zorganizowane w sposób iteracyjny czy rekurencyjny.

3.1. Rozwiązywanie równania kwadratowego

Zajmiemy się tutaj algorytmizacją zadania, które chyba najczęściej jest rozwiązywane na lekcjach matematyki — rozwiązywaniem równania kwadratowego. Występuje ono również dość często wśród zadań, dla których pisze się programy na lekcjach informatyki. Skorzystamy z najczęściej stosowanych wzorów na pierwiastki tego równania, czyli z metody, którą niektórzy nazywają algorytmem „delty”. Wykorzystamy również wzory Viète’a. W tym drugim przypadku utworzymy algorytm, który jest poprawny dla komputerowych obliczeń — uzasadnienie tego faktu nie jest jednak całkiem proste.

Równanie kwadratowe podaje się na ogół w następującej postaci:

$$ax^2 + bx + c = 0 \quad (3.1)$$

a więc danymi w tym zadaniu są trzy liczby: a , b i c — współczynniki trójmianu. Zakłada się przy tym, że współczynnik a jest różny od zera, gdyż w przeciwnym razie pierwszy składnik w tym równaniu byłby równy zeru i stałoby się ono równaniem pierwszego stopnia. Pierwiastki równania (3.1) wyznacza się w algorytmie „delty” za pomocą następujących wzorów:

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a}, \quad x_2 = \frac{-b + \sqrt{\Delta}}{2a} \quad (3.2)$$

gdzie $\Delta = b^2 - 4ac$. Kiedy potrafimy obliczyć pierwiastki równania z tych wzorów? Odpowiedź jest dość prosta — wtedy, gdy można wykonać wszystkie działania występujące w tych wzorach. Dwa z tych działań są obwarowane warunkami — dzielić można tylko przez liczbę różną od zera i pierwiastek kwadratowy istnieje, gdy liczba podpierwiastkowa nie jest ujemna. W przypadku wzorów (3.2) dzielenie przez $2a$ jest zawsze wykonalne, gdyż założyliśmy, że w równaniu (3.1) współczynnik a jest różny od zera. Natomiast aby obliczenie wartości pierwiastka kwadratowego było możliwe, wartość wyróżnika Δ musi

być nieujemna. Zatem jeśli $\Delta < 0$, to równanie nie ma pierwiastków (a dokładniej — nie ma pierwiastków rzeczywistych). Ponadto, jeśli $\Delta = 0$, to $x_1 = x_2$, czyli oba pierwiastki, są sobie równe. Na podstawie tych rozważań możemy podać kolejne kroki algorytmu obliczania pierwiastków równania kwadratowego, gdy dane są jego współczynniki.

Algorytm rozwiązywania równania kwadratowego

Dane: Współczynniki a , b i c równania (3.1).

Wyniki: Pierwiastki równania (3.1), jeśli dane współczynniki rzeczywiście określają równanie kwadratowe i równanie ma pierwiastki. Jeśli równanie nie ma pierwiastków, to wypisz odpowiedni komunikat.

Krok 1. Jeśli $a = 0$, to wypisz komunikat, że nie jest to równanie kwadratowe i zakończ algorytm.

Krok 2. Oblicz wartość wyróżnika $\Delta = b^2 - 4ac$.

Krok 3. Jeśli $\Delta < 0$, to wypisz komunikat, że równanie kwadratowe nie ma pierwiastków i zakończ algorytm.

Krok 4. Jeśli $\Delta = 0$, to oblicz oba pierwiastki z tego samego wzoru: $x_1 = x_2 = -b/2a$, wypisz ich wartości i zakończ algorytm.

Krok 5. {W tym przypadku $\Delta > 0$.} Oblicz pierwiastki x_1 i x_2 ze wzorów (3.2), wypisz ich wartości i zakończ algorytm. ■

Podany algorytm^[11] jest najczęściej wykorzystywanym algorytmem rozwiązywania równania kwadratowego zarówno na lekcjach matematyki, jak i informatyki. Dla niektórych wartości współczynników może on dawać jednak bardzo niedokładne wartości pierwiastków.

T

Rozważmy przykład zaczerpnięty z książki [EI-I], w którym współczynniki równania (3.1) mają następujące wartości podane z czterema cyframi znaczącymi: $a = 1$, $b = -6.433$ i $c = 0.009474$, i wykonajmy obliczenia, posługując algorytmem „delty” i przeprowadzając obliczenia z dokładnością do 4 cyfr znaczących. Oznacza to, że każdy wynik pośredni będzie zaokrąglany do czterech cyfr. Zaokrąglenie do czterech cyfr polega na obcięciu cyfr, począwszy od piątej cyfry, i dodanie 1 do czwartej cyfry, gdy piąta cyfra (czyli pierwsza odrzucona) jest większa lub równa 5.

Otrzymujemy z obliczeń: wartość wyróżnika $\Delta = 41.34$ oraz wartości pierwiastków $x_1 = 0.0015$ i $x_2 = 6.43$. W porównaniu z dokładną wartością pierwszego pierwiastka x_1 , która wynosi 0.001473, otrzymany wynik ma tylko

pierwszą taką samą cyfrę znaczącą, natomiast dla drugiego pierwiastka x_2 , którego dokładna wartość wynosi 6.431, otrzymany wynik jest zgodny na trzech cyfrach znaczących.

Jaka jest przyczyna takiej niedokładności w obliczeniach wartości pierwiastka x_1 ? Na lekcjach matematyki na ogół nie wykonujemy obliczeń dla wartości współczynników, dla których otrzymuje się tak niedokładne rozwiązanie. Oczywiście, wartości podane w przykładzie zostały tak dobrane, by pokazać, że obliczenia komputerowe nie zawsze gwarantują otrzymanie dokładnych wyników. Powodów utraty dokładności może być wiele — nie będziemy o nich tutaj dyskutować, gdyż sformułowanie odpowiednich wniosków wykracza poza zakres szkolnej matematyki. Chcemy jedynie wskazać na przykładzie tego prostego zadania obliczeniowego, że znalezienie dokładnego rozwiązania może stanowić dla komputerów pewien problem. Wynika stąd wniosek: niektóre obliczenia komputerowe wykonywane według matematycznych wzorów wymagają analizy, zanim zapiszemy je w postaci programu komputerowego.

W przypadku rozwiązywania równania kwadratowego powód utraty dokładności wartości pierwiastków jest dość prosty do wytłumaczenia. Musimy jedynie uzmysłowić sobie, że wszystkie obliczenia w komputerze są wykonywane na liczbach, które mają ograniczoną liczbę cyfr znaczących. W naszych obliczeniach przyjęliśmy dla uproszczenia, że są one wykonywane z dokładnością do czterech cyfr znaczących, by nie wykonywać działań na liczbach o wielu cyfrach. Przypatrzmy się teraz, skąd się wzięła niedokładność w obliczeniach. Wartość iloczynu $4ac = 0.03790$ jest mała w stosunku do wartości $b^2 = 41.38$. Oznacza to, że wartość wyróżnika $\Delta = 41.34$ jest w przybliżeniu równa b^2 , a więc wartość pierwiastka kwadratowego, $\sqrt{\Delta} = 6.430$,

jest bliska bezwzględnej wartości b , równej 6.433. Jeśli więc współczynnik b jest ujemny, to przy obliczaniu wartości pierwiastka x_1 są odejmowane dwie bliskie sobie liczby — w naszym przykładzie odejmowaliśmy 6.430 od 6.433. (Jeśli współczynnik b jest dodatni, to przy obliczaniu x_2 są odejmowane dwie bliskie sobie liczby). I to jest właśnie powodem utraty dokładności w algorytmie „delty” rozwiązywania równania kwadratowego. Zauważmy jeszcze, że w obu przypadkach mniej dokładny jest tylko jeden z pierwiastków.

Algorytm, który wykazuje opisaną wyżej własność dla niektórych danych, nazywa się **algorytmem niestabilnym**. W takim algorytmie błędy zaokrągleń wyników pośrednich mogą wpływać na niedokładność wyników końcowych. ■

Na lekcjach matematyki w szkole są podawane również inne wzory, które można zastosować do obliczania pierwiastków równania kwadratowego i które pomagają usunąć opisaną wyżej niedogodność, pojawiającą się przy stosowaniu wzorów (3.2) — są nimi **wzory Viète’a**. Wyrażają one sumę oraz iloczyn

pierwiastków równania kwadratowego. Załóżmy, jak dotychczas, że współczynnik a w równaniu (3.1) jest różny od zera, a więc, że jest to rzeczywiście równanie kwadratowe. Wtedy mamy:

$$x_1 + x_2 = \frac{-b}{a} \quad (3.3)$$

$$x_1 x_2 = \frac{c}{a}$$

Skorzystamy z drugiego ze wzorów (3.3), wyrażającego iloczyn pierwiastków. Jak już zauważyliśmy, tylko jeden z pierwiastków może być niedokładnie obliczany przy stosowaniu wzorów (3.2). Zatem nasze zmodyfikowane postępowanie będzie polegało na: obliczeniu bardziej dokładnej wartości pierwiastka za pomocą wzorów (3.2) i obliczeniu drugiego pierwiastka ze wzoru Viète'a (3.3) na iloczyn pierwiastków. Wybieramy ten z pierwiastków we wzorach (3.2), w którym w liczniku są dodawane dwie liczby o tych samych znakach, gdyż — jak wyjaśniliśmy powyżej — niedokładność bierze się z odejmowania bliskich sobie liczb. Ten wybór zależy od znaku współczynnika b — jeśli $b < 0$, to wybieramy x_2 , a w przeciwnym przypadku wybieramy x_1 . Szczegółowy opis tego algorytmu pozostawiamy do samodzielnego wykonania. Można się przy tym posłużyć programem w języku Pascal, który zamieszczamy poniżej.

Ćwiczenie 3.1. Opisz w postaci listy kroków algorytm rozwiązywania równania kwadratowego (opisanego nieformalnie powyżej), w którym do obliczania pierwiastków tego równania korzysta się z jednego ze wzorów (3.2) i drugiego wzoru Viète'a (3.3). ■

Opis algorytmu z ćwiczenia 3.1. w języku Python jest zamieszczony poniżej, a opis w języku Pascal pozostawiamy do samodzielnego wykonania.. W tej funkcji mamy trzy zagłębiające się instrukcje warunkowe — najpierw jest sprawdzane, czy jest to równanie kwadratowe, jeśli tak, to głębiej jest sprawdzane, czy wyróżnik delta nie jest ujemny, a dopiero w najgłębszej instrukcji warunkowej są obliczane wartości pierwiastków w zależności od znaku współczynnika b .



```
import math

def Rowkwad(a,b,c):
    print("Rozwiązanie równania kwadratowego")
    if a == 0:
        print("To nie jest równanie kwadratowe")
    else:
```



```

delta=b*b-4*a*c
if delta < 0:
    print("Równanie nie ma pierwiastków rzeczywistych")
else:
    print("Pierwiastki równania kwadratowego:")
    delta=math.sqrt(delta)
    if b < 0:
        x2=(-b+delta)/(2*a)
        print("x1 =", c/(a*x2), "x2 =", x2)
    else:
        x1=(-b-delta)/(2*a)
        print("x1 =", x1, "x2 =", c/(a*x1))

```



Czytelnikom zainteresowanym wnikliwszymi rozważaniami na temat dokładności obliczania pierwiastków równania kwadratowego polecamy podręcznik [EI-I, punkt 6.4.1] oraz program ROWNKWAD.EXE dołączony do poradnika [EI-III]. Program ten ilustruje opisaną powyżej utratę dokładności obliczeń dla różnej liczby cyfr znaczących w danych i w wynikach. ■

3.2. Rozwiązywanie równania liniowego

Jednym z najprostszych zadań geometrycznych jest wyznaczanie punktów, w których prosta przecina osie układu współrzędnych. Ogólne równanie prostej, zwane również **równaniem liniowym**, ma następującą postać:

$$ax + by = c \quad (3.4)$$

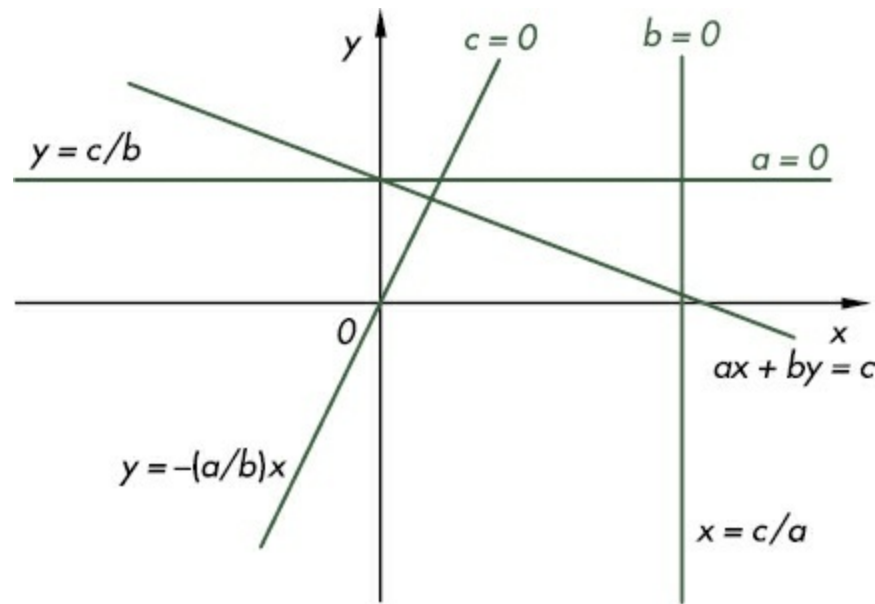
Zatem danymi w tym zadaniu są trzy liczby: a , b i c — współczynniki prostej. Geometrycznie, prosta w układzie współrzędnych może być równoległa do jednej z osi lub nie być równoległa do żadnej z osi. Ponieważ danymi w naszym zadaniu są trzy liczby, kolejne kroki odpowiedzi na pytanie o położenie prostej w układzie współrzędnych należy uzależnić od ich wartości, a dokładniej od tego, czy są to liczby różne od zera czy równe zeru.

Założmy na początku, że wszystkie współczynniki są różne od zera. Aby otrzymać punkt przecięcia prostej o równaniu (3.4) z osią Ox , przyjmujemy $y = 0$, gdyż na tej osi wszystkie punkty mają taką współrzędną rzędną. Zatem w tym przypadku równanie (3.4) ma postać $ax = c$ i stąd otrzymujemy punkt $(c/a, 0)$. Podobnie otrzymujemy punkt $(0, c/b)$ na osi Oy .

Teraz założmy, że dokładnie jeden współczynnik w równaniu (3.4) jest równy zeru. Jeśli $a = 0$, to $y = c/b$ i to dla wszystkich wartości x . Zatem w tym przypadku prosta przechodzi przez punkt $(0, c/b)$ na osi rzędnych i jest równoległa do osi odciętych Ox . Podobnie, jeśli $b = 0$, to prosta (3.4) jest równoległa do osi rzędnych i przechodzi przez punkt $(c/a, 0)$ na osi Ox . Jeśli natomiast $c = 0$, to równanie przyjmuje postać:

$$ax = -by$$

i prosta przecina osie współrzędnych w początku układu, czyli w punkcie $(0, 0)$. Te cztery przypadki są zilustrowane na rysunku 3.1.



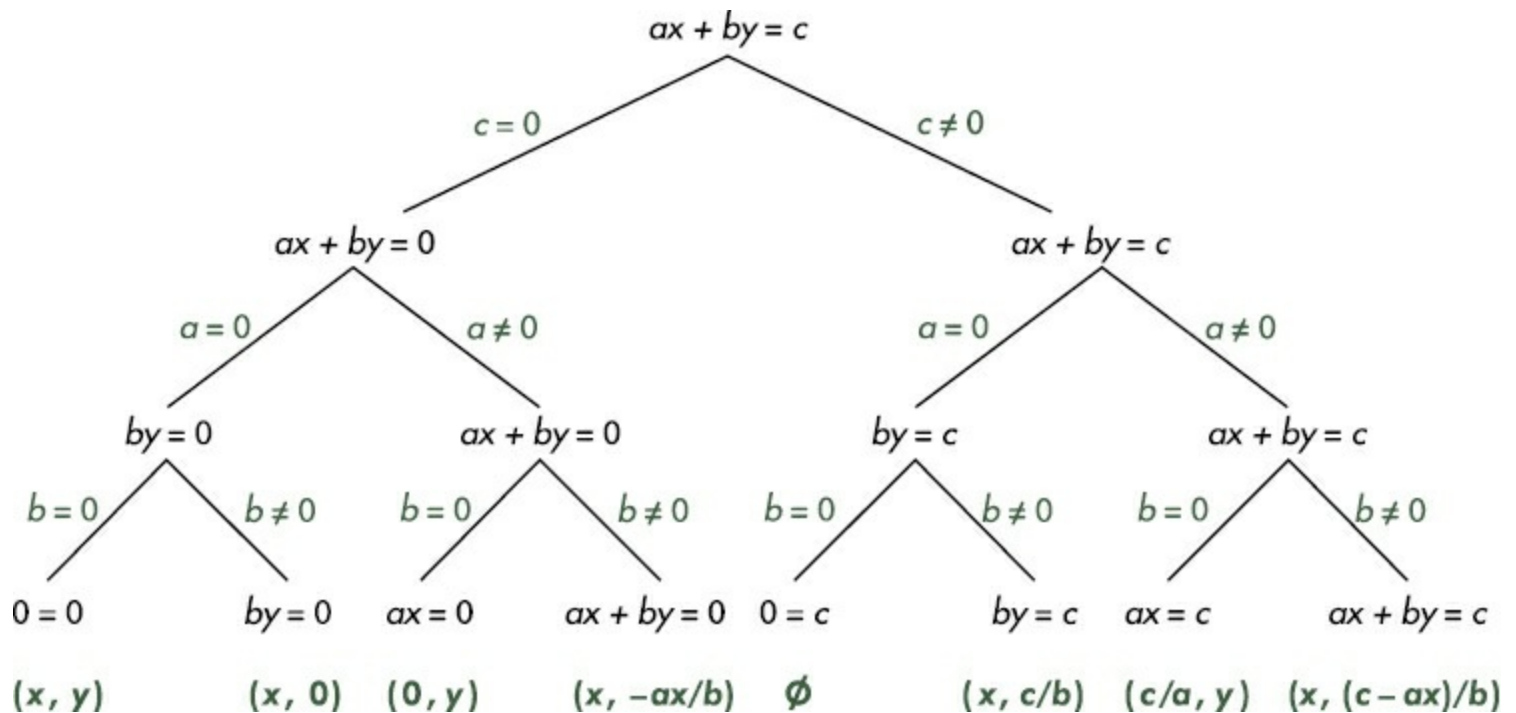
Rysunek 3.1. Różne położenia prostej o równaniu (3.4) w układzie współrzędnych, gdy co najwyżej jeden ze współczynników jest równy zeru

Rozważmy teraz przypadek, gdy dwa współczynniki równania (3.4) są równe zeru. Jeśli $a = b = 0$, to otrzymujemy równanie $c = 0$, które jest spełnione tylko wtedy, gdy $c = 0$. A więc jeśli w tym przypadku c nie jest równe 0, to układ jest sprzeczny. Jeśli $a = c = 0$, to równanie (3.4) ma postać $by = 0$, zatem jest spełnione przez każdą parę liczb $(x, y) = (x, 0)$ dla dowolnego x . Jeśli w tym przypadku ponadto $b = 0$, to równanie (3.4) jest spełnione przez każdą parę liczb (x, y) . Podobnie analizujemy przypadek $b = c = 0$.

Liczba podprzypadków w tym na pozór prostym zadaniu jest dość duża. W opisie algorytmu wyznaczania położenia prostej o równaniu (3.4) w układzie współrzędnych na podstawie wartości trzech dowolnych liczb będących współczynnikami tej prostej musimy uwzględniać wszystkie możliwości. Jak zapewnić, żeby żadnego z możliwych przypadków nie opuścić?

W dalszej części tego punktu uprościmy nasze zadanie i będziemy analizować jedynie postać równania (3.4) w zależności od wartości współczynników, a do postawionego na początku pytania o punkty przecięcia się prostej opisanej tym równaniem z osiami układu współrzędnego wrócimy w ćwiczeniu 3.3.

Podamy teraz inny sposób generowania przypadków — został on schematycznie zobrazowany na rysunku 3.2 w postaci drzewa. Wszystkie rozważane wyżej przypadki można podzielić na dwie grupy: w jednej $c = 0$, a w drugiej $c \neq 0$. Pierwszej grupie przypadków odpowiada w drzewie lewe poddrzewo, a drugiej — prawe. Na następnym poziomie drzewa, czyli dla ustalonej wartości c , rozważamy współczynnik a , dla którego również $a = 0$ lub $a \neq 0$, czyli lewe poddrzewo rozgałęzia się na dwa poddrzewa w zależności od wartości a . Następny poziom w drzewie odpowiada wartościom trzeciego współczynnika. Podobnie powstaje prawe poddrzewo.



Rysunek 3.2. Drzewo wszystkich możliwych podprzypadków postaci równania (3.4) w zależności od wartości współczynników

Jaką mamy pewność, że żaden przypadek nie został pominięty? Poniżej wierzchołków końcowych drzewa jest zapisana postać równania oraz zbiór rozwiązań w postaci pary współrzędnych, które spełniają równanie w przypadku kończącym się w tym wierzchołku (zmienne x i y występujące w tych parach mogą przyjmować dowolne wartości rzeczywiste). Łatwo sprawdzić, że wierzchołki końcowe odpowiadają wszystkim możliwym układom zerowych współczynników równania (3.4): $a = b = c = 0$, $a = b = 0$, $a = c = 0$, $b = c = 0$, $a = 0$, $b = 0$, $c = 0$ i przypadkowi, gdy żaden współczynnik nie jest zerem. To nam gwarantuje, że jeśli tylko algorytm będzie przebiegał wszystkie gałęzie tego drzewa, to żaden przypadek nie zostanie pominięty.

Zapisując powyższy algorytm w postaci listy kroków, musimy bardzo uważać na zachowanie odpowiedniej kolejności kroków.

Algorytm analizowania postaci równania liniowego (3.4)

Dane: Współczynniki a , b i c równania (3.4).

Wynik: Odpowiednia postać równania (3.4), w zależności od tego, który współczynnik jest równy zeru lub różny od zera.

Krok 1. Jeśli $c = 0$, to przejdź do kroku 9.

Krok 2. Jeśli $a = 0$, to przejdź do kroku 6.

Krok 3. Jeśli $b = 0$, to przejdź do kroku 5.

Krok 4. {Żaden współczynnik nie jest równy 0.} Równanie (3.4) ma ogólną postać $ax + by = c$. Zakończ algorytm.

Krok 5. {Przypadek $b = 0$.} Równanie (3.4) przyjmuje postać $ax = c$. Zakończ algorytm.

Krok 6. Jeśli $b = 0$, to przejdź do kroku 8.

Krok 7. {Przypadek $a = 0$.} Równanie (3.4) przyjmuje postać $by = c$. Zakończ algorytm.

Krok 8. {Przypadek $a = b = 0$.} Równanie (3.4) przyjmuje postać $c = 0$ dla $c \neq 0$, jest więc sprzeczne. Zakończ algorytm.

Krok 9. Jeśli $a = 0$, to przejdź do kroku 13.

Krok 10. Jeśli $b = 0$, to przejdź do kroku 12.

Krok 11. {Przypadek $c = 0$.} Równanie (3.4) przyjmuje postać $ax + by = 0$. Zakończ algorytm.

Krok 12. {Przypadek $b = c = 0$.} Równanie (3.4) przyjmuje postać $ax = 0$, jest więc spełnione przez wszystkie pary liczb $(0, y)$, czyli przez punkty leżące na osi Oy . Zakończ algorytm.

Krok 13. Jeśli $b = 0$, to przejdź do kroku 15.

Krok 14. {Przypadek $a = c = 0$.} Równanie (3.4) przyjmuje postać $by = 0$, jest więc spełnione przez wszystkie pary liczb $(x, 0)$, czyli przez punkty leżące na osi Ox . Zakończ algorytm.

Krok 15. {Przypadek $a = b = c = 0$.} Równanie (3.4) jest spełnione przez wszystkie pary liczb (x, y) . Zakończ algorytm. ■

Ten opis analizy postaci równania (3.4) w zależności od wartości współczynników, chociaż bardzo dokładny, jest mało czytelny. By nie pomylić przypadków, niektóre kroki algorytmu zostały opatrzone komentarzami w nawiasach, zawierającymi odpowiednie warunki spełniane przez współczynniki równania. Zauważmy, że w każdym kroku, który kończy analizę równania, trzeba wpisać polecenie zakończenia działania algorytmu, bowiem w przeciwnym razie po wykonaniu tego kroku obliczenia byłyby kontynuowane od następnego kroku.

Ćwiczenie 3.2. Narysuj schemat blokowy, realizujący powyższy opis algorytmu,

bardziej „czytelny” niż jego słowny opis. Swoim „wyglądem” powinien przypominać drzewo z rysunku 3.2. ■

Ćwiczenie 3.3. Zmodyfikuj opis algorytmu analizującego postać równania (3.4) oraz jego schemat blokowy wykonany w ćwiczenia 3.2 tak, aby wynikiem działania algorytmu były punkty przecięcia się wykresu funkcji liniowej opisanej równaniem (3.4) z osiami współrzędnych. ■

3.3. Rozwiązywanie układu równań liniowych

Rozważymy teraz rozwiązywanie układu dwóch równań liniowych. Przyjmijmy, że ten układ ma następującą postać:

$$\begin{cases} ax + by = c \\ dx + ey = f \end{cases} \quad (3.5)$$

Rozwiązywanie układu równań (3.5) jest niczym innym, jak szukaniem takiego punktu (x, y) , dla którego każde z równań jest spełnione. Jest więc to punkt, przez który przechodzą obie proste opisane układem równań (3.5), czyli punkt ich przecięcia. Wiadomo, że dwie proste przecinają się, gdy nie są równoległe.

Szczególnym przypadkiem równoległości dwóch prostych jest ich pokrywanie się — wówczas można powiedzieć, że każdy punkt jednej prostej jest punktem przecięcia z drugą prostą. Układ równań (3.5) nie powinien więc mieć rozwiązania, gdy proste są równoległe.

Układ ten rozwiązuje się na ogół w następujący sposób: najpierw wyznacza się na przykład zmienną x z pierwszego równania i wstawia do drugiego równania. W ten sposób drugie równanie zawiera już tylko jedną niewiadomą y . Następnie tak wyznaczone y wstawia się do równania na x . Po wykonaniu odpowiednich przekształceń wzory na rozwiązanie układu (3.5) można zapisać w następującej postaci:

$$\begin{cases} x = \frac{ce - bf}{w} \\ y = \frac{af - cd}{w} \end{cases} \quad (3.6)$$

gdzie $w = ae - db$. Wielkość w nazywa się **wyznacznikiem** układu równań (3.6). Z postaci wzorów na x i y wynika, że układ ma rozwiązanie, gdy wyznacznik w jest różny od zera. Z kolei równość $w = 0$ jest spełniona, gdy $ae = db$, czyli gdy $a/b = d/e$. Zauważmy, że te ilorazy stanowią współczynniki

kierunkowe prostych tworzących układ równań (3.5).

Sposób wyznaczenia wartości x i y jako rozwiązań układu równań (3.5) jest szczególnym przypadkiem metody znajdowania rozwiązań układów równań liniowych o dowolnej liczbie niewiadomych, zwanej metodą **eliminacji Gaussa**. Wzory (3.6) to szczególna postać **wzorów Cramera**, wyrażających rozwiązanie układu równań liniowych poprzez wartości odpowiednich wyznaczników układu.

Zatem równość tych współczynników oznacza, że te proste są równoległe. Rozwiązując algebraicznie układ równań (3.5), otrzymaliśmy więc warunek, o którym wspomnieliśmy w interpretacji geometrycznej rozwiązania tego układu.

Zapisanie algorytmu wyznaczania rozwiązania układu równań (3.5) w postaci listy kroków, schematu blokowego lub w języku programowania pominiemy. Zwrócimy natomiast uwagę na dość ważną i, niestety, niekorzystną własność rozwiązań układu równań (3.5) i w tym celu poprosimy Cię o wykorzystanie arkusza kalkulacyjnego, który bywa przydatny przy analizowaniu i rozwiązywaniu niektórych problemów algorytmicznych.

Ćwiczenie 3.4. Znajdź rozwiązanie układu równań (3.5) dane wzorami (3.6) dla wartości współczynników $a = 3$, $b = -5$, $c = -2$, $d = 1$, $e = 6$, $f = 7$. Posłuż się w tym celu arkuszem kalkulacyjnym. Narysuj również w arkuszu proste (3.5) dla tych współczynników. Narysuj również proste (3.5) w przypadku, gdy wyznacznik układu jest równy 0. ■

Odnotujmy, że wartość wyznacznika dla danych z ćwiczenia 3.4 wynosi $w = 23$, a rozwiązaniem jest punkt $(1, 1)$.

Zachęcamy teraz bardzo do wykonania ćwiczenia dla danych zaczerpniętych z podręcznika [EI-I, punkt 6.4.2].

Ćwiczenie 3.5. Wykonaj podobne obliczenia jak w ćwiczeniu 3.3 dla danych $a = 3.000$, $b = 4.127$, $c = 15.41$, $d = 1.000$, $e = 1.374$, $f = 5.147$. W szczególności zanotuj wartość wyznacznika dla tych danych i przedstaw wyniki w postaci graficznej. ■

To nie jest błąd, że dla zestawu danych z ćwiczenia 3.5 widzisz wykres tylko jednej prostej. Algorytm na pewno działa poprawnie, gdyż dla zestawu danych z ćwiczenia 3.4 pojawiły się wykresy dwóch prostych. Czy domyślasz się, co się stało? Wybierając rysowanie tylko jednej prostej, raz jednej, a raz drugiej, można zauważyć, że niemal się pokrywają. Nie są jednak równoległe, gdyż zostało znalezione rozwiązanie układu $x = 13.666$ i $y = -6.2$.

Aby podać dokładniejsze wyjaśnienie, zauważmy że wartość wyznacznika układu dla danych z ostatniego ćwiczenia $w = -0.005$, jest więc bardzo mała w porównaniu z wartością współczynników układu. Pamiętamy, że jeśli wyznacznik jest równy zeru, to proste są równoległe, więc w szczególności mogą się pokrywać. Zatem jeśli wartość wyznacznika jest bliska zeru, to oznacza, że

proste są niemal równoległe. Gdy są onerysowane na ekranie grubą linią, wówczas mogą się pokrywać i właśnie to jest powodem pojawienia się wykresu tylko jednej prostej w tym przypadku.

Wyjaśnienie powyżej ma charakter geometryczny. Faktyczny powód pojawienia się jednej prostej jest związany z błędami występującymi przy dzieleniu przez stosunkowo małą wartość wyznacznika. Podobnie jak w rozwiązywaniu równania kwadratowego za pomocą tradycyjnych wzorów (3.2) opisujących jego pierwiastki, również tutaj dochodzi do utraty dokładności obliczeń, spowodowanej dzieleniem dość dużej liczby przez stosunkowo małą liczbę. Przy rozwiązywaniu równania kwadratowego można zapobiec wystąpieniu błędów zaokrągleń, posługując się dodatkowo wzorami Viète'a. Niestety, rozwiązywanie układu równań liniowych jest problemem, dla którego nie można podać algorytmu odpornego na błędy zaokrągleń. Jest to bowiem **problem źle uwarunkowany, niestabilny**, czyli niedokładność wyników nie jest związana z algorytmem rozwiązywania, ale z samym problemem — każdy algorytm rozwiązywania układu równań (3.5) dla danych z ćwiczenia 3.5 da wynik „rozmyty” — nie może bowiem zmienić kąta między dwiema prostymi, który w przypadku danych z tego ćwiczenia jest bardzo mały. W sposób nieformalny można powiedzieć, że to, czy widać punkt przecięcia się blisko siebie leżących prostych, zależy od grubości ołówka, którym rysujemy. ■



Bardziej szczegółowe rozważania na temat dokładności rozwiązywania układu równań liniowych znajdziesz w podręczniku [EI-I, punkt 6.4.2]. Ponadto w materiałach dołączonych do poradnika [EI-III] znajduje się program demonstracyjny UKLLIN.EXE, który ilustruje opisaną utratę dokładności obliczeń podczas rozwiązywania układu równań liniowych z różną dokładnością. ■

3.4. Zadania

Zadanie 3.1. Zbuduj schemat blokowy algorytmu obliczania pola trójkąta S na podstawie **wzoru Herona**, w którym występują jedynie długości boków a , b i c trójkąta:

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

gdzie $p = (a + b + c)/2$. Nie zapomnij o sprawdzaniu warunku, czy liczba podpierwiastkowa nie jest mniejsza od zera. Jakie jest znaczenie warunku, czy

np. $p - a > 0$? Czy jest dopuszczalne, by $S = 0$?

Napisz program w języku Pascal lub Python służący do obliczania wartości S .

Pole trójkąta można również obliczać ze wzoru, w którym występują długości: boku i wysokości na niego opuszczonej. Dlaczego, stosując ten wzór, nie trzeba sprawdzać żadnego warunku?

Powinieneś umieć:

- ▶ **zabezpieczać się** w algorytmach **przed wykonywaniem zabronionych działań** przez sprawdzanie odpowiednich warunków;
- ▶ **wyjaśnić, skąd mogą się brać niedokładności** w obliczeniach komputerowych — ich źródłem jest najczęściej wykonywanie obliczeń na liczbach, których różnica wartości bezwzględnych jest bardzo duża lub bardzo mała;
- ▶ **podać przykłady obliczeń komputerowych, które mogą być niedokładne, wskazać źródło tych niedokładności** i umieć zaradzić tym trudnościom wówczas, gdy jest to możliwe;
- ▶ **radzić sobie z uwzględnianiem w obliczeniach wielu przypadków**, na które rozgałęzia się algorytm rozwiązania.

[\[1\]](#) Chociaż rozważania od następnego akapitu są nieco trudniejsze, zachęcamy jednak do przynajmniej pobieżnego zapoznania się z nimi.

Rozdział 4. Porządkowanie kilku liczb

Tu poznasz:

- sposoby **porządkowania kilku liczb**: dwóch, trzech, czterech i pięciu;
- sposoby **wykorzystania drzewa** do planowania obliczeń.

W tym rozdziale zaczynamy rozważać jedno z najważniejszych i najpopularniejszych zagadnień informatycznych — **porządkowanie**, w informatyce zwane najczęściej **sortowaniem**. O znaczeniu problemu porządkowania i jego algorytmów jest mowa na początku rozdziału 6. Problem porządkowania definiuje się następująco:

Problem porządkowania (sortowania)

Dane: Liczba naturalna n i ciąg n liczb x_1, x_2, \dots, x_n .

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej. ■

Ponieważ algorytmy porządkowania służą do znajdowania uporządkowania liczb względem ich wartości, wykonujemy w nich operacje porównania i — gdy jest to konieczne ze względu na złe uporządkowanie porównywanych liczb — zmieniamy ich kolejność, czyli przestawiamy je. Porównanie w algorytmach porządkowania oznacza zbadanie prawdziwości jednej z nierówności: słabej (\leq lub \geq) albo ostrej ($<$ lub $>$).

Ćwiczenie 4.1. Przedstaw, w wybranej przez siebie reprezentacji, algorytm porządkowania dwóch liczb a i b . ■

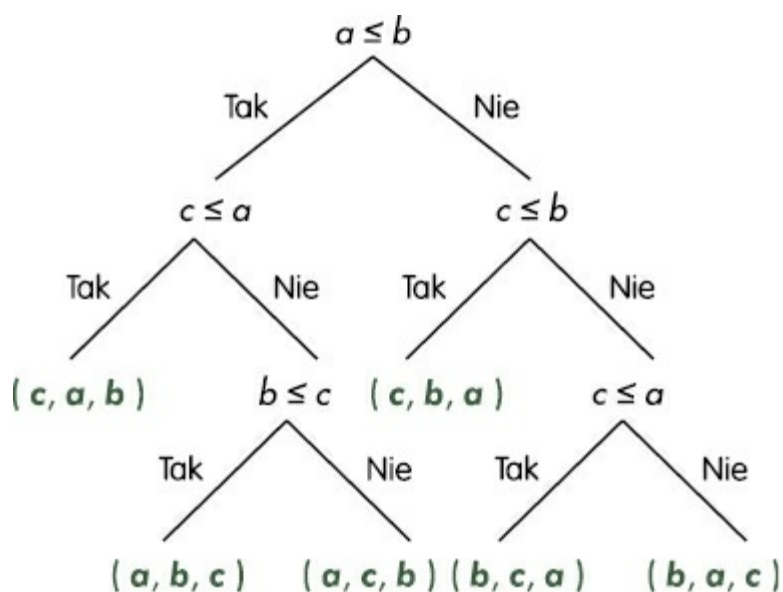
4.1. Porządkowanie trzech liczb

W algorytmie podanym jako rozwiązanie ćwiczenia 4.1 wykonujesz zapewne jedno porównanie, by ustawić dwie liczby, najpierw mniejszą, a potem większą z nich. Dla dwóch liczb trudno jest nawet podać algorytm, według którego wykonywalibyśmy więcej niż jedno porównanie.

Mając do uporządkowania trzy liczby, możemy postąpić następująco: najpierw ustawić a i b w odpowiedniej kolejności, a potem wstawić c w odpowiednie miejsce względem a i b . W pierwszym etapie wykonujemy jedno porównanie, a w drugim? Załóżmy, że po pierwszym porównaniu, czy $a \leq b$, otrzymujemy uporządkowanie (a, b) . Wtedy jeśli $c \leq a$, to (c, a, b) jest szukaną kolejnością. W przeciwnym razie musimy jeszcze sprawdzić, czy $c \leq b$. Jeśli tak, to otrzymujemy uporządkowanie (a, c, b) , a w przeciwnym razie (a, b, c) . Podobnie postępujemy, gdy po pierwszym porównaniu kolejność liczb a i b jest następująca (b, a) . Wszystkie te przypadki można zapisać w postaci drzewa

przedstawionego na rysunku 4.1.

Ćwiczenie 4.2. Opisz w postaci listy kroków algorytm porządkowania trzech liczb, którego drzewo jest przedstawione na rysunku 4.1. ■



Rysunek 4.1. Drzewo algorytmu porządkującego trzy liczby

Ćwiczenie 4.3. Przerób drzewo algorytmu z rysunku 4.1 na schemat blokowy, który czyta trzy liczby a , b , c i drukuje te liczby w porządku od najmniejszej do największej. Następnie zapisz ten algorytm w języku Pascal lub Python i wykonaj obliczenia dla różnych danych, by się przekonać, że właściwie je porządkuje. Uwzględnij dane, w których niektóre albo wszystkie liczby są sobie równe. ■

Posłużymy się teraz drzewem algorytmu przedstawionym na rysunku 4.1 do intuicyjnej interpretacji niektórych fragmentów takich drzew, związanej zwłaszcza z liczbą wykonywanych operacji w algorytmie oraz sprawdzaniem poprawności jego działania.

Zacznijmy od tej drugiej kwestii, czyli od uzasadnienia, że algorytm zapisany w postaci tego drzewa rzeczywiście porządkuje poprawnie trzy dowolne liczby. Jeśli mamy do uporządkowania trzy liczby: a , b i c , to w zależności od ich wartości w wyniku porządkowania możemy otrzymać każdą z możliwych ich kolejności. Ile jest takich kolejności? A więc, na ile sposobów można ustawić trzy liczby? Łatwo to wyprowadzić. Jeśli dwie liczby można ustawić na dwa sposoby, jako (a, b) i (b, a) , to wszystkie uporządkowania trzech liczb można otrzymać, umieszczając trzecią liczbę c w różnych miejscach w stosunku do tych dwóch liczb, w każdym z tych dwóch uporządkowań. Dla każdego uporządkowania można to zrobić na trzy sposoby, wstawiając c : na początku, w środku i na końcu. Dla każdego z dwóch uporządkowań dwóch liczb otrzymujemy trzy uporządkowania trzech liczb, czyli w sumie jest ich 6. Zauważmy, że $6 = 1 \cdot 2 \cdot 3$.

Zatem wszystkich możliwych uporządkowań trzech liczb, czyli możliwych odpowiedzi algorytmu porządkującego trzy liczby, jest 6 i wszystkie znajdują się w **liściach** (wierzchołkach końcowych) drzewa algorytmu przedstawionego na rysunku 4.1. Ponadto sprawdzane nierówności właściwie kierują do odpowiednich liści. Można więc powiedzieć, że algorytm opisany tym drzewem poprawnie porządkuje trzy liczby.

Z rozważań obok wynika, że w najmniej korzystnym (zwanym także najgorszym) przypadku danych algorytm reprezentowany w postaci drzewa algorytmu jest tym lepszy (szybszy), im „niższe” jest (lub ma mniejszą wysokość) to drzewo. Ten intuicyjny wniosek można wykorzystać do uzasadnienia, że niektóre algorytmy są najszybsze wśród możliwych — zobacz binarne umieszczanie (punkt 9.2) oraz problem porządkowania (punkt 10.4). Z tych rozważań wynika również, że przedstawione w tym rozdziale algorytmy porządkowania kilku liczb wykonują swoje zadanie za pomocą najmniejszej możliwej liczby porównań! Są to więc **algorytmy optymalne** pod względem złożoności obliczeniowej.

Zastanówmy się teraz, ile porównań wykonujemy w algorytmie zapisanym w drzewie dla poszczególnych układów danych. Aby to obliczyć, zauważmy, że liczba porównań prowadzących do wyróżnionej odpowiedzi algorytmu jest równa liczbie wierzchołków pośrednich na drodze z korzenia do liścia zawierającego tę odpowiedź — tę liczbę będziemy nazywali **długością** takiej drogi. Zatem wyniki (c, a, b) i (c, b, a) otrzymujemy po wykonaniu dwóch porównań, a pozostałe — po trzech. Największa długość drogi z korzenia do wierzchołka końcowego, nazywana **wysokością drzewa** algorytmu, jest więc największą liczbą operacji wykonywanych w algorytmie dla jakiegokolwiek możliwego układu danych. Wielkość ta nazywa się **złożonością obliczeniową** lub **pracochłonnością algorytmu**. Jak widać na przykładzie algorytmu porządkowania trzech liczb, jest to **złożoność pesymistyczna**, czasem mówimy **złożoność najgorszego przypadku danych** — w niektórych przypadkach bowiem algorytm może działać szybciej.

4.2. Porządkowanie czterech liczb

Ćwiczenie 4.4. Powtórz rozumowanie z poprzedniego punktu i oblicz, ile istnieje różnych uporządkowań czterech liczb. ■

Ćwiczenie 4.5. Wzorując się na algorytmie z poprzedniego punktu, przedstaw w postaci drzewa algorytm porządkowania czterech liczb a, b, c i d . Jaka jest jego złożoność? ■

Uporządkowanie czterech liczb a, b, c i d można znaleźć w dwóch krokach:

1) uporządkować najpierw trzy liczby a, b i c algorytmem z poprzedniego punktu,

2) umieścić czwartą liczbę d w uporządkowanym ciągu trzech liczb.

Czy taki podałeś algorytm jako rozwiązanie ćwiczenia 4.5? Drzewo dla tego algorytmu można otrzymać z drzewa przedstawionego na rysunku 4.1 przez dorysowanie w każdym liściu poddrzewa, odpowiadającego umieszczeniu elementu d w ciągu znajdującym się w tym wierzchołku.

Ćwiczenie 4.6. Utwórz drzewo algorytmu porządkowania czterech liczb, opisanego powyżej. O ile wzrosła wysokość drzewa, czyli ile wykonujesz porównań, by wśród trzech uporządkowanych liczb umieścić czwartą liczbę zgodnie z ich porządkiem? ■

Zasugerowany obok sposób wstawiania czwartej liczby do trójki liczb uporządkowanych jest szczególnym przypadkiem algorytmu binarnego umieszczania — zobacz punkt 9.2.

Jeśli wysokość wzrosła o 3, to zastanów się jeszcze raz. Odpowiedź, której tutaj oczekujemy, wynosi 2 i odpowiada takiemu sposobowi umieszczania, w którym liczba d jest najpierw porównywana ze środkową liczbą z trzech liczb uporządkowanych. Przekonaj się, że otrzymane w ten sposób drzewo algorytmu porządkowania czterech liczb ma wysokość 5, a więc dowolne cztery liczby można uporządkować, wykonując co najwyżej 5 porównań.

4.3. Porządkowanie pięciu liczb

Zadanie 4.1. Powtórz jeszcze raz rozumowanie z poprzednich punktów i oblicz, ile istnieje różnych uporządkowań pięciu liczb. ■

Ćwiczenie 4.7. Korzystając z doświadczenia zdobytego przy porządkowaniu trzech i czterech liczb, zaproponuj algorytm porządkowania pięciu liczb a , b , c , d i e . ■

Rozszerzenie algorytmu porządkowania czterech liczb na porządkowanie pięciu liczb w taki sposób, w jaki otrzymaliśmy ten pierwszy algorytm w poprzednim punkcie, daje algorytm, w którym wykonuje się 8 porównań. Możliwy jest jednak algorytm, który wykonuje co najwyżej 7 takich operacji — dalsze rozważania poświęcimy jego wyprowadzeniu.

Wszystkich możliwych porównań między pięcioma liczbami jest tyle, ile par dwóch różnych liczb można wybrać spośród pięciu liczb. Obliczmy to. Pierwszą liczbę możemy połączyć w cztery pary z czterema pozostałymi liczbami. Drugą — już tylko z trzema: trzecią, czwartą i piątą, gdyż para z pierwszą liczbą została uwzględniona przy liczeniu par zawierających pierwszą liczbę. Z tego samego powodu trzecią liczbę można połączyć z dwoma, a czwartą — tylko z piątą liczbą. Wszystkie pary z piątą liczbą były już uwzględnione przy rozważaniu par z poprzednimi liczbami. A więc wszystkich różnych

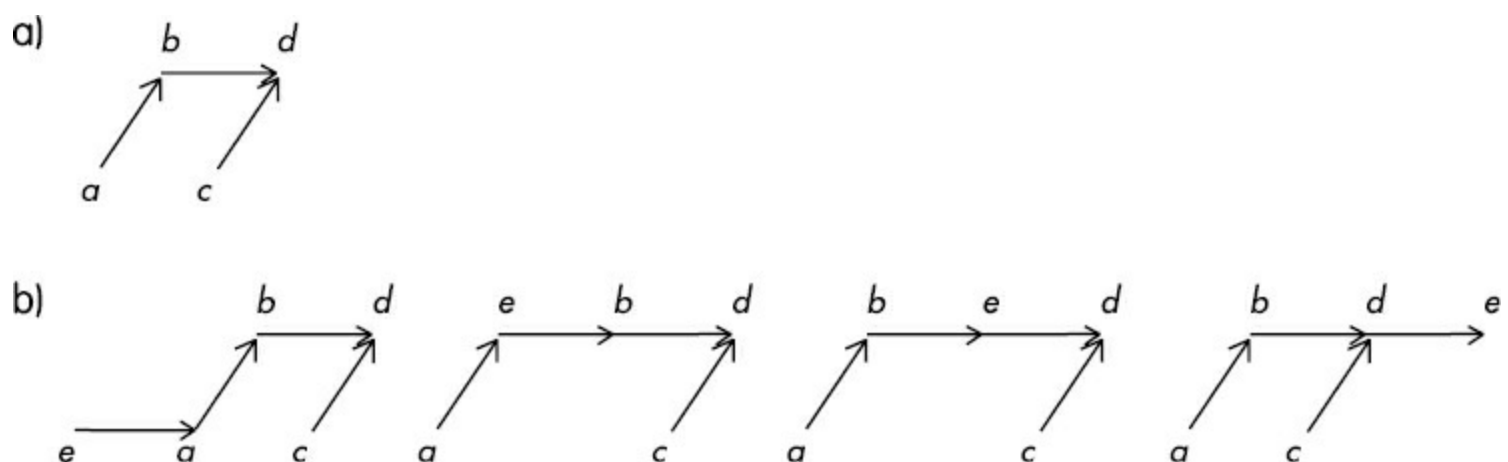
nieuporządkowanych par spośród pięciu liczb jest $4 + 3 + 2 + 1$, czyli 10.

Ile najwięcej porównań wykonujesz, posługując się algorytmem z ćwiczenia 4.7? Nie powinieneś więcej niż 10, gdyż — jak wykazaliśmy — wszystkich możliwych porównań między pięcioma liczbami jest właśnie tyle. Jeśli nie 10, to o ile mniej?

Algorytm porządkowania pięciu liczb nie jest tak oczywisty, jak porządkowanie trzech czy czterech liczb. Zaproponował go H. B. Demuth w swojej pracy doktorskiej z 1956 roku.

Jeśli chcemy, by algorytm wykonywał mniej niż 10 porównań, to pewne nierówności między porządkowanymi liczbami muszą wynikać ze sprawdzonych już nierówności. Jest to możliwe na przykład wtedy, gdy: po wykonaniu porównań i otrzymaniu nierówności $a \leq b \leq c$ nic nie wiemy o związku d z tymi trzema liczbami — i w tej sytuacji sprawdzamy nierówność $d \leq b$. Niezależnie od odpowiedzi zaoszczędzimy jedno porównanie: jeśli bowiem $d \leq b$, to również $d \leq c$, a jeśli $d > b$, to również $d > a$.

Podamy teraz, w postaci listy kroków, algorytm porządkowania pięciu liczb, w którym wykonuje się co najwyżej 7 porównań dzięki temu, że wyniki co najmniej trzech takich operacji z dziesięciu możliwych uzyskuje się na podstawie wyników porównań już wykonanych. Efekty wykonywania kolejnych kroków algorytmu można śledzić na diagramach zamieszczonych na rysunku 4.2, gdzie liczba x (która może być jedną z porządkowanych liczb a, b, c, d lub e) jest mniejsza od liczby y (jednej z liczb a, b, c, d lub e) wtedy i tylko wtedy, gdy istnieje droga skierowana wzdłuż strzałek z x do y .



Rysunek 4.2. Podprzypadki z algorytmu porządkowania pięciu liczb a, b, c, d i e algorytmem, w którym w najgorszym razie wykonuje się siedem porównań. Na tych diagramach liczba x jest mniejsza od liczby y wtedy i tylko wtedy, gdy z x istnieje droga skierowana wzdłuż strzałek do y

Algorytm porządkowania pięciu liczb

Dane: Pięć liczb: a, b, c, d i e .

Wynik: Znaleźć uporządkowanie tych liczb od najmniejszej do największej.

Krok 1. Porównaj a z b oraz c z d . Załóżmy, że spełnione są nierówności $a \leq b$ i $c \leq d$. Jeśli jest inaczej, to zamień odpowiednie liczby miejscami.

Krok 2. Porównaj b z d . Załóżmy, że wynik jest $b \leq d$. Jeśli jest inaczej, to zamień miejscami b z d oraz a z c . {Zobacz rysunek 4.2a.}

Krok 3. Wstaw e we właściwe miejsce między liczby a , b i d . Do tego są potrzebne dwa porównania. Najpierw porównaj e z b i w zależności od wyniku — jeśli $e < b$, to porównaj a z e , a jeśli $b \leq e$, to porównaj e z d . {Na rysunku 4.2b są przedstawione wszystkie możliwe układy liczb po wykonaniu tego kroku.}

Krok 4. Wstaw c w odpowiednie miejsce między elementy mniejsze niż d . Można to wykonać za pomocą dwóch dodatkowych porównań: najpierw c z liczbą środkową wśród mniejszych od d , a następnie — z lewą lub prawą, w zależności od wyniku pierwszego porównania. ■

Sposób wstawiania liczby e w kroku 3. oraz liczby c w kroku 4. jest również szczególnym przypadkiem algorytmu binarnego umieszczania — zobacz punkt 9.2.

Zgodnie z tym algorytmem wykonuje się nie więcej niż 7 porównań: dwa w kroku 1., jedno w kroku 2., dwa w kroku 3. i co najwyżej dwa w kroku 4.

T

Przedstawimy teraz bardziej szczegółowe uwagi dotyczące automatycznego porządkowania pięciu liczb za pomocą opisanego algorytmu. Przede wszystkim określmy, w jakiej kolejności należy pamiętać dane liczby, aby cztery podprzypadki w kroku 4. można było opisać jednym podprogramem. Oznaczmy ciąg pięciu porządkowanych liczb przez $(x_1, x_2, x_3, x_4, x_5)$. W kroku 1. sprawdzamy, czy $x_1 \leq x_2$ oraz $x_4 \leq x_5$, i jeśli któraś z tych nierówności nie jest spełniona, to zamieniamy odpowiednie elementy miejscami. W kroku 2. sprawdzamy nierówność $x_2 \leq x_5$ i jeśli nie jest ona spełniona, to zamieniamy miejscami x_2 z x_5 oraz x_1 z x_4 . W tym momencie w ciągu x elementy z algorytmu są przyporządkowane następująco: $(x_1, x_2, x_3, x_4, x_5) = (a, b, e, c, d)$. W następnym kroku umieszczamy e , czyli x_3 , na odpowiednim miejscu. Zgodnie z algorytmem, sprawdzamy najpierw, czy $x_3 < x_2$. Jeśli tak, to następnie sprawdzamy, czy $x_3 < x_1$. Jeśli tak, to mamy pierwszą sytuację z rysunku 4.2b — wstawiamy e na początek ciągu i otrzymujemy uporządkowanie (e, a, b, c, d) , a w przeciwnym razie zamieniamy tylko miejscami e z b . Jeśli natomiast $x_3 \geq x_2$, to sprawdzamy nierówność $x_3 \leq x_5$ (czyli $e \leq d$). Jeśli jest spełniona, to pozostawiamy ciąg bez zmian, w przeciwnym razie zamieniamy e i d miejscami. Ta ostatnia zamiana może wydać się dziwna, gdyż $c \leq d$, tymczasem w ciągu

mamy odwrotną kolejność tych elementów — jest to uzasadnione tym, że chcemy, aby element c , umieszczany w następnym kroku, był na tym samym miejscu w ciągu x we wszystkich przypadkach. Dokładniej, czterem przypadkom na rysunku 4.2b odpowiadają teraz następujące układy liczb w ciągu x : (e, a, b, c, d) , (d, e, b, c, d) , (a, b, e, c, d) , (a, b, d, c, e) .

Pozostał do wykonania ostatni krok algorytmu, polegający na umieszczeniu liczby c , która zajmuje to samo czwarte miejsce w ciągu, w odpowiednim miejscu podciągu, który składa się z trzech pierwszych elementów ciągu x . Zauważmy tutaj, że „zepsuta” w poprzednim kroku (ostatni przypadek) kolejność elementów c i d będzie „naprawiona” w tym kroku. Realizację tego kroku zaczynamy od sprawdzenia nierówności $x_4 < x_2$, a następnie sprawdzamy albo $x_4 < x_1$, albo $x_4 < x_3$. ■

Opisany powyżej sposób wykonania algorytmu ilustruje, że niezbędnym składnikiem realizacji algorytmu (na przykład w postaci programu komputerowego) są struktury danych. Najczęściej cytuje się w tym kontekście sformułowanie: *Algorytmy + Struktury danych = Programy*, zaczerpnięte z tytułu jednej z książek Niklausa Wirtha, twórcy języka programowania Pascal [Wirth].

Powyższa analiza działania algorytmu, z uwzględnieniem sposobu pamiętania w nim danych, jest w tym przypadku niezbędnym etapem na drodze od opisu algorytmu (słownego lub w postaci listy kroków) do jego realizacji w postaci komputerowej. Dokładniej, musieliśmy określić **strukturę danych**, w której będą pamiętane dane, wyniki pośrednie i wynik końcowy, oraz podać, w jaki sposób są wykonywane na niej poszczególne kroki algorytmu. Komputerową realizację algorytmu można bowiem podać dopiero wtedy, gdy są znane algorytm, struktury danych, na których on operuje, oraz sposób jego realizacji na tych strukturach.

Po opisaniu algorytmu i struktur danych, na których ma działać, jesteśmy gotowi do przedstawienia realizacji algorytmu porządkowania pięciu liczb w postaci programu komputerowego.

Poniższa procedura w języku Pascal jest dokładnym zapisem opisanej wyżej realizacji algorytmu porządkowania pięciu liczb, w której jest wykonywanych 7 porównań. Porządkowane liczby są dane w tablicy $x[1..5]$ i w niej pozostają po uporządkowaniu.



```
procedure SortPiec(var x: Tablica5);  
  procedure Przetaw(var u, w: integer);  
    {Procedura przestawia miejscami liczby u i w.}
```

```

    var z: integer;
begin
    z := u; u := w; w := z
end; {Przestaw}

procedure Umiesc;
    {Realizacja kroku 4. - procedura umieszcza c = x[4]
    w odpowiednim miejscu wsrod elementow x[1..3].}
    var c: integer;
begin
    c := x[4];
    if c < x[2] then begin
        x[4] := x[3]; x[3] := x[2];
        if c < x[1] then begin x[2] := x[1]; x[1] := c end
        else x[2] := c
    end {c < x[2]}
    else {Przypadek x[2] <= c}
        if c < x[3] then begin x[4] := x[3]; x[3] := c end
        {W przeciwnym razie, c jest na swoim miejscu.}
    end; {Umiesc}
var e: integer;
begin {Program glowny: SortPiec.}
    if x[1] > x[2] then Przestaw(x[1], x[2]);
    if x[4] > x[5] then Przestaw(x[4], x[5]);
    if x[2] > x[5] then begin
        Przestaw(x[2], x[5]); Przestaw(x[1], x[4])
    end; {Koniec kroku 2.}
    e := x[3];
    if e < x[2] then begin
        x[3] := x[2];
        if e < x[1] then begin x[2] := x[1]; x[1] := e end
        else x[2] := e
    end
    else if e > x[5] then Przestaw(x[3], x[5]);
        {Koniec kroku 3.}
    Umiesc {Krok 4.}
end; {SortPiec}

```

Ćwiczenie 4.8. Przekonaj się, że procedura SortPiec rzeczywiście wykonuje nie

więcej niż 7 porównań, bez względu na porządek między liczbami danymi do uporządkowania. Policz również, ile zamian elementów porządkowanego ciągu pięciu liczb jest wykonywanych w tej procedurze. ■

Ćwiczenie 4.9. Jeśli znasz język Python, to „przepisz” procedurę SortPiec na funkcję w tym języku. Zacznij od dokładnej analizy struktur danych, jakich użyjesz. ■

4.4. Zadania i problemy

Problem 4.1. Wykaż, że wśród n liczb można wybrać $n(n - 1)/2$ różnych par nieuporządkowanych.

Zadanie 4.2. Jak wynika z podanego opisu, w algorytmie porządkowania pięciu liczb wykonuje się co najwyżej 7 porównań. Postaraj się uzasadnić, w których krokach tego algorytmu oszczędza się w sumie 3 porównania z 10 możliwych do wykonania między pięcioma liczbami.

Powinieneś umieć:

- **porządkować** szybko **kilka elementów**, dwa, trzy, cztery i pięć;

oraz wiedzieć:

- w jaki sposób **drzewa algorytmów mogą modelować algorytmy** porządkowania i jak mogą być wykorzystywane do planowania obliczeń;

- że nieodłączną częścią komputerowej realizacji algorytmu są **struktury danych**, czyli sposoby pamiętania informacji w algorytmie oraz wykonywania na nich operacji.

Rozdział 5. O czym mówią dane — algorytmy iteracyjne

Tu poznasz:

- ▶ **algorytmy działające na zbiorach** o dowolnej liczbie danych;
- ▶ sposób reprezentowania **zbiorów w algorytmach**;
- ▶ rolę **wartownika zbioru**;
- ▶ **algorytmy przeszukiwania zbiorów** w poszukiwaniu elementu;
- ▶ sposoby znajdowania podstawowych **statystyk zbioru**: elementów największego i najmniejszego, średniej, mediany i modalnej;
- ▶ sprawiedliwy **sposób rozgrywania turnieju** tenisowego;
- ▶ **algorytm znajdowania lidera** w zbiorze.

Algorytmy przedstawione w poprzednich rozdziałach wykonują kilka porównań lub są obliczeniami według kilku wzorów. Chociaż algorytmy te są uniwersalne w tym sensie, że danymi w nich mogą być dowolne liczby, to jednak ich ilość jest z góry ustalona. W tym rozdziale po raz pierwszy pojawiają się algorytmy, w których wykonuje się obliczenia na zbiorach elementów. Powinny one działać poprawnie zarówno na zbiorze złożonym z 1, 2 lub 3 elementów, jak i na zbiorach o 100 czy 10000 elementów.

Zaczynamy tutaj zajmować się problemami, do których rozwiązywania komputery są niezbędne, potwierdzając tym przewidywania Ady z pierwszej połowy XIX wieku, że znaczenie tych maszyn będzie wynikać z możliwości wielokrotnego wykonywania przez nie tej samej lub nieco zmienionej operacji.

Wielkości, które będziemy wyznaczać dla zbioru danych, zależą zwykle od wartości wszystkich danych w tym zbiorze. Ich określenie musi więc być związane z przejrzaniem całego zbioru i wykonaniem pewnych działań, na ogół takich samych, na wszystkich elementach tego zbioru. W algorytmach odpowiada temu **iteracja**, czyli **powtarzanie** pewnych operacji lub nawet całych fragmentów obliczeń (np. kilku kroków).

W najprostszym przypadku elementami zbioru mogą być liczby. Będą nas jednak interesować również zbiory utworzone z kart do gry, uczniów jakiejś klasy, zawodników zgłoszonych do udziału w turnieju rozgrywek indywidualnych itd. Elementom w różnych zbiorach można na ogół przyporządkować liczby (cechy), np. kartom — wartości, uczniom — wzrost lub imiona, zawodnikom — ich kategorię lub poziom gry. Przyjmujemy więc, że elementy w zbiorze można scharakteryzować liczbami lub przynajmniej o każdym dwóch elementach można powiedzieć, czy są takie same lub który z nich jest większy — można

zatem je porównywać. Dalej na ogół mówimy o zbiorach elementów, nazywanych również danymi, jak o zbiorach liczb, chociaż czasem posługujemy się językiem którejs z interpretacji tych liczb. Najczęściej elementy zbiorów utożsamiamy z ich wartościami, zwłaszcza wtedy, gdy tymi elementami są liczby.

Jakie informacje chcemy uzyskać na podstawie analizy posiadanego zbioru elementów? Jakie informacje są często podawane na podstawie dużego zbioru [\[1\]](#) danych? Przypomnijmy sobie informacje prasowe i dane z urzędów statystycznych. Jeśli dane w zbiorze powtarzają się, to ciekawa może być informacja o ich częstości. Ważna jest również rozpiętość danych, czyli różnica między największą a najmniejszą liczbą występującą wśród danych. Osobną grupę wielkości, określanych dla zbiorów danych, stanowią tzw. miary centralności (tendencji centralnej), do których należą średnia, mediana i modalna. Przypomnimy, jak te wielkości są definiowane.

Dla każdej danej w zbiorze można określić jej **częstość**, tj. ile razy powtarza się w tym zbiorze. **Rozpiętością** zbioru jest różnica między największą a najmniejszą liczbą występującą w tym zbiorze — najmniejszą liczbę w zbiorze nazywamy również **minimum** zbioru, a największą — **maksimum** zbioru. **Miarami centralności** danych w zbiorze, zwanymi również **statystykami** zbioru, są: średnia, mediana i moda (wartość modalna). **Średnia** jest równa sumie wszystkich danych podzielonej przez ich liczbę. Średnia, tak samo jak mediana, nie musi być równa żadnej z danych w zbiorze. **Mediana** jest środkową wartością zbioru w tym sensie, że jeśli zbiór uporządkujemy, to występuje ona w środku tego uporządkowania. Na przykład, medianą zbioru {3, 2, 1, 2, 4} jest liczba 2, gdyż po uporządkowaniu tego zbioru mamy (1, 2, 2, 3, 4). W przypadku parzystej liczby danych jest ona równa średniej z $n/2$ oraz $n/2 + 1$ wartości. Na przykład, medianą zbioru {2, 1, 5, 4} jest liczba 3, gdyż po uporządkowaniu tego zbioru mamy (1, 2, 4, 5), z czego bierzemy średnią z liczb 2 oraz 4. **Moda** zaś jest najczęściej występującą daną w zbiorze. Zbiór może nie mieć modalnej, gdy żadna dana nie występuje w nim częściej niż jeden raz, albo może mieć jedną lub wiele modalnych.

Ćwiczenie 5.1. Wśród danych może występować liczba **odstająca** od pozostałych. Na przykład w wykazie nieobecności wszystkich uczniów liczba dni nieobecności jednego z nich, przewlekłe chorego, jest zapewne znacznie większa od liczby nieobecności pozostałych uczniów. Która z miar centralności jest w takim przypadku najbardziej odpowiednia do określenia przeciętnej nieobecności wszystkich uczniów? ■

Ćwiczenie 5.2. Jak przyjęliśmy, dane nie muszą być zbiorem liczb. Jeśli dysponujesz listą najpopularniejszych wśród uczniów utworów muzycznych, to która z miar jest odpowiednia do określenia centralności takich danych, czyli znalezienia najbardziej popularnego utworu? Jaka byłaby Twoja odpowiedź,

gdyby można było te dane uporządkować, np. alfabetycznie? ■

W kolejnych punktach tego rozdziału najpierw omówimy, w jaki sposób zbiory są reprezentowane i przeszukiwane w algorytmach, a następnie przedstawimy algorytmy znajdowania zdefiniowanych wyżej wielkości.

5.1. Reprezentowanie i przeszukiwanie zbioru

Podobnie jak w przypadku każdego innego obiektu występującego w algorytmach, tak i w przypadku zbioru, najpierw chcemy wiedzieć, jak można go reprezentować, a później — jak wykonywać na nim podstawowe operacje. Taką podstawową operacją w odniesieniu do zbiorów jest ich przeszukiwanie.

5.1.1. Reprezentowanie zbioru danych w algorytmach

Pojawia się pierwsze pytanie, w jaki sposób reprezentować w algorytmach zbiory o nieustalonej z góry liczbie elementów. To pytanie jest o tyle ważne, że we wszystkich obliczeniach na zbiorach, a ogólniej — w obliczeniach powtarzających się, iteracyjnych — musimy mieć pewność, że:

- ▶ z jednej strony — żaden element zbioru nie został pominięty przez algorytm,
- ▶ a z drugiej — obliczenia nie będą się powtarzać w nieskończoność.

Zbiór danych, dla którego wyznaczamy w algorytmie pewne wielkości lub na których wykonujemy wybrane operacje, może być:

- ▶ czytany z klawiatury albo z pliku,
- ▶ znajduje się w tablicy (np. w przypadku programów w języku Pascal) lub w liście (w przypadku programów w języku Python)[\[2\]](#).

Dalej korzystamy głównie z tego drugiego sposobu dostępu do danych w algorytmach.

Uwagi obok są kolejnym potwierdzeniem, że opis algorytmu powinien być poprzedzony dyskusją i ustaleniem, w jaki sposób są przechowywane dane. Jak się wielokrotnie przekonamy, od sposobu reprezentowania danych zależy postać algorytmu.

W obu przypadkach zbiór elementów występujący w algorytmie jest dany jako ciąg elementów. Jest to istotne założenie. Jak wiadomo z matematyki, elementy w zbiorze nie występują w jakiejś wyróżnionej kolejności — każda jest możliwa. Jeśli algorytm działa na zbiorze, to kolejność elementów tego zbioru może być również dowolna, jednak ze względu na konieczność zapisania zbioru w postaci określonej struktury danych (takiej np. jak tablica lub lista) przyjmujemy, że dla

ustalonej w dowolny sposób kolejności jego elementów zbiór jest reprezentowany w postaci ciągu swoich elementów.

Bez względu na sposób reprezentowania zbioru, powinniśmy wiedzieć, ile zawiera on elementów, czyli jaka jest jego **moc**. Można wyróżnić dwa sposoby określenia zbioru, a przez to również jego mocy:

1. Pierwszy sposób polega na tym, że na początku ciągu danych podajemy, ile elementów zawiera zbiór, a następnie wymieniamy jego elementy. Wówczas wiemy dokładnie, ile należy wykonać działań na wszystkich elementach zbioru.
2. W drugim sposobie natomiast na końcu zbioru umieszczamy wyróżniony element, który nie należy do tego zbioru — ten element nazywa się **wartownikiem**, gdyż w algorytmach jego rolą jest „pilnowanie”, by żadne obliczenia dotyczące tego zbioru nie wykroczyły poza jego elementy. Dla uproszczenia będziemy na ogół zakładać, że nasze zbiory danych są utworzone z liczb dodatnich, zatem wartownikiem może być jakakolwiek liczba ujemna, np. -1 .

Rolę wartownika w obliczeniach, nawet nieco ogólniejszą niż tylko zakończenie zbioru, ilustrujemy w następnym punkcie, gdzie omawiamy jedną z najprostszych operacji wykonywanych na zbiorach. Następnie, na przykładzie obliczania średniej, można się przekonać, że czasem występują niewielkie różnice w algorytmach wykorzystujących jedną i drugą reprezentację zbioru. A w dalszych fragmentach książki reprezentację zbioru dobieramy w zależności od postawionego problemu i operacji wykonywanych w algorytmie.

5.1.2. Liniowe przeszukiwanie zbioru

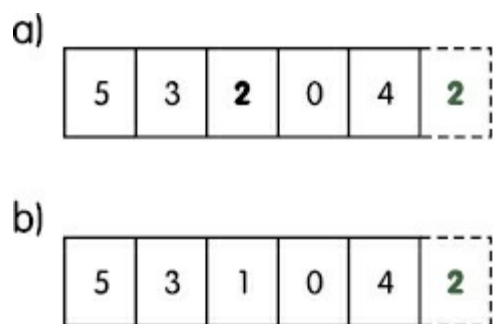
Jedno z najprostszych pytań, jakie można postawić w odniesieniu do zbioru danych, brzmi: czy zawiera on poszukiwany przez nas element. Jeśli nie mamy żadnych dodatkowych informacji o elementach rozważanego zbioru, np. czy nie są one uporządkowane od najmniejszego do największego, to najprostszą metodą udzielenia odpowiedzi na to pytanie jest sprawdzenie element po elemencie (w kolejności, w jakiej są one podawane lub przechowywane), czy któryś z nich jest równy poszukiwanemu elementowi. Taki sposób przeglądania zbioru nazywa się **przeszukiwaniem liniowym**.

Podczas liniowego przeszukiwania zbioru danych reprezentowanych w ciągu, porównując kolejne elementy zbioru z poszukiwanym elementem, musimy również sprawdzać, czy nie wyczerpaliśmy już całego zbioru. Postępujemy tak bez względu na to, czy liczba elementów jest podana na początku zbioru czy kończy się on wartownikiem. Proponujemy samodzielnie opisać tę wersję przeszukiwania liniowego.

Ćwiczenie 5.3. Przedstaw, w wybranej przez siebie reprezentacji, algorytm

przeszukiwania liniowego, w którym jest sprawdzany warunek, czy przeszukiwanie osiągnęło już koniec ciągu danych. ■

Omówimy teraz szczegółowo algorytm przeszukiwania liniowego, który jest bardzo „elegancki” i ma dobre własności informatyczne. Wykorzystamy w tym celu wartownika, ale w trochę innym sensie niż przedstawionym powyżej. Jeśli y jest poszukiwanym elementem zbioru, to dołączamy do tego zbioru jeden nowy element, równy właśnie y — zobacz rysunek 5.1. Jaki jest efekt przeszukiwania tak rozbudowanego zbioru w poszukiwaniu elementu y ? Otóż zawsze zakończy się ono znalezieniem elementu równego y . Na końcu należy tylko sprawdzić, czy znaleziony element y znajduje się w ciągu, czy też wystąpił na dołączonej pozycji. W pierwszym przypadku badany zbiór zawiera element y , a w drugim — y nie należy do tego 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.



Rysunek 5.1. Ciąg pięciu elementów z dołączonym wartownikiem dla algorytmu przeszukiwania liniowego: a) poszukiwany element $y = 2$ należy do zbioru; b) nie należy do zbioru

Jeszcze jedna uwaga o postaci rozwiązania problemu przeszukiwania zbioru w poszukiwaniu wyróżnionego elementu. Możemy poszukiwać elementu o danej wartości (jak w tym przykładzie) lub elementu o wyróżnionej własności (np. najmniejszego). W zależności od zastosowań, jeśli zbiór zawiera wyróżniony element, to celem przeszukiwania może być znalezienie tej wyróżnionej wartości elementu (np. chcemy tylko wiedzieć, jaki jest najmniejszy wzrost uczniów w szkole) lub wskazanie miejsca znalezionego elementu w zbiorze (czyli kto jest najniższym uczniem — wtedy również wiemy, jakiego wzrostu jest najniższy uczeń w szkole). W tym drugim przypadku otrzymujemy zatem bogatszą informację. W dalszej części książki będziemy postępować w zależności od konkretnej sytuacji, na ogół jednak zależy nam na znalezieniu tej bogatszej informacji o poszukiwanym elemencie.

Zamieńmy teraz słowny opis algorytmu przeszukiwania na opis w postaci listy kroków.

Algorytm przeszukiwania liniowego z wartownikiem

Dane: Zbiór elementów dany w postaci ciągu; y — poszukiwany element.

Wynik: Jeśli y należy do zbioru, to podaj jego miejsce w ciągu, a w przeciwnym razie sygnalizuj brak takiego elementu w zbiorze.

Krok 1. {Utworzenie wartownika na końcu ciągu.} Utwórz na końcu ciągu wartownika i umieść w nim element y .

Krok 2. Dla kolejnego elementu z w danym ciągu, jeśli $z = y$, to przejdź do kroku 3., w przeciwnym razie powtórz ten krok.

Krok 3. Jeśli z jest wartownikiem, to zbiór nie zawiera elementu y , w przeciwnym razie — miejsce elementu z wskazuje pierwsze wystąpienie elementu w ciągu. ■

W najgorszym przypadku, stosując algorytm przeszukiwania liniowego, musimy przejrzeć cały zbiór (ciąg), aby sprawdzić istnienie w nim danego elementu. Wówczas liczba wykonywanych porównań jest o jeden większa od liczby elementów w zbiorze. Jeśli zbiór zawiera poszukiwany element, to zwykle liczba tych porównań jest mniejsza (zobacz również ostatni akapit w tym punkcie).

Znalezienie jakiejś rzeczy trwa na ogół krócej niż stwierdzenie jej braku. Przypomnij sobie tylko, gdy ostatnim razem szukałeś czegoś w domu i nie znalazłeś.

Zauważmy, że rola wartownika w tym algorytmie jest nieco odmienna (i nieco ogólniejsza) od roli omówionej wcześniej. Tutaj wartownik (a dokładniej jego wartość) zmienia się wraz ze zmianą poszukiwanego elementu, natomiast w reprezentacji zbioru jest to element, który służy jedynie do odróżniania końca zbioru od jego elementów.

Podamy jeszcze wersję algorytmu przeszukiwania liniowego, w której ciąg znajduje się w tablicy. W podobny sposób opisujemy algorytm przeszukiwania binarnego (punkt 9.2).

Algorytm przeszukiwania liniowego z wartownikiem — wersja z tablicą

Dane: Zbiór elementów dany w tablicy $a[k..l]$, gdzie $k \leq l$; y — poszukiwany element.

Wynik: Takie s ($k \leq s \leq l$), że $a_s = y$, lub przyjąć $s = -1$, jeśli $y \neq a_i$, dla każdego i ($k \leq i \leq l$).

Krok 1. $a_{l+1} := y$ {Utworzenie wartownika na końcu ciągu.}

Krok 2. Dla $s = k, \dots, l, l+1$, jeśli $a_s = y$, to przejdź do kroku 3.

Krok 3. Jeśli $s = l+1$, to zbiór nie zawiera elementu y , więc przyjmij $s := -1$, w przeciwnym razie mamy $a_s = y$. ■

Poniższe programy są zapisem algorytmu przeszukiwania liniowego z

wartownikiem w językach Pascal i Python. Zwróćmy uwagę, że realizacją kroku 2. z algorytmu jest jedna instrukcja iteracyjna while.



```
function PrzeszukiwanieLiniowe(a:TablicaIn; k,l,y:integer):integer;  
  var s:integer;  
begin  
  a[l+1] := y;  
  s := k;  
  while y <> a[s] do s := s + 1;  
  if s <= l then  
    PrzeszukiwanieLiniowe := s  
  else PrzeszukiwanieLiniowe := -1  
end; {PrzeszukiwanieLiniowe}
```



```
def szukaj_z_wartownikiem(x, lista):  
  lista = lista + [x]  
  i = 0  
  while lista[i] != x:  
    i = i + 1  
  if i < len(lista)-1:  
    print("lista[",i, "] =",x)  
  else:  
    print("lista nie zawiera elementu", x)
```

Skomentujmy jeszcze, że w języku Python przeszukiwanie liniowe może być wykonane z wykorzystaniem wbudowanych możliwości tego języka, w szczególności można posłużyć się operatorem in, odszukiwaniem indeksu poszukiwanego elementu (list.index) oraz wyjątkiem (try i except), gdy takiego elementu nie ma w liście – zobacz funkcje poniżej. Warto jednak pamiętać, że za jednokrotnym posłużeniem się operatorem in stoi... liniowy algorytm przeszukiwania listy lista, a więc obie realizacje przeszukiwania mają taką samą złożoność.



```
def szukaj_z_in(x, lista):  
  if x in lista:
```



```

    print("x[",lista.index(x),"] =",x)
else:
    print("lista nie zawiera elementu", x)
def szukaj_try(x, lista):
    try:
        print("x[",lista.index(x),"] =",x)
    except:
        print("lista nie zawiera elementu", x)

```

Przeszukiwanie liniowe zbioru elementów można usprawnić, gdy mamy dodatkowe informacje o tym zbiorze, np. jeśli wiadomo, że jest on uporządkowany (zobacz problem 5.2 oraz w punkcie 10.1 — zastosowanie takiego przeszukiwania w algorytmie porządkowania). Kolejnym usprawnieniem przeszukiwania liniowego jest przyjęcie innej metody poruszania się po zbiorze, np. przez połowienie (zobacz punkt 9.2).

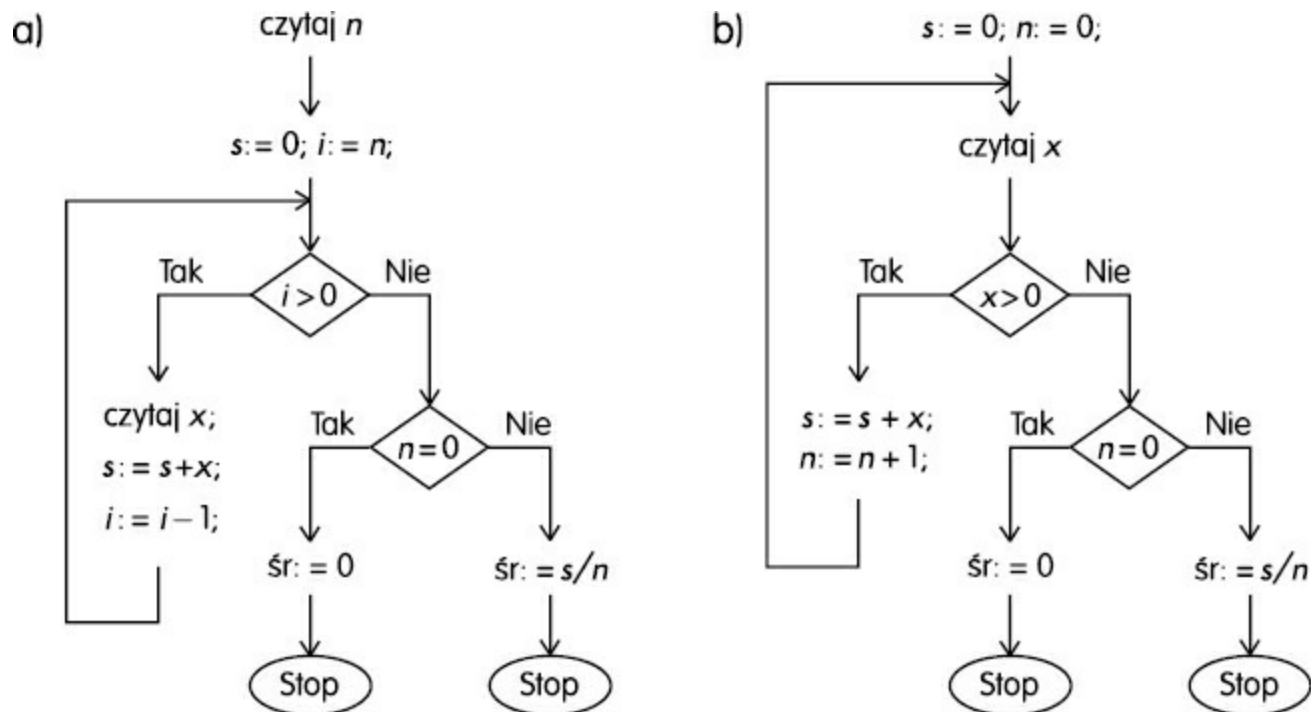
5.2. Obliczanie średniej

Średnia ze zbioru liczb jest równa sumie jego elementów podzielonej przez ich liczbę. Na rysunku 5.2 są przedstawione klasyczne schematy blokowe algorytmów obliczania wartości średniej z liczb podawanych np. z klawiatury. W schemacie 5.2a pierwszą wczytaną liczbą jest liczba elementów w zbiorze, a w schemacie 5.2b — zbiór danych liczb (dodatnich) jest zakończony wartownikiem (liczbą niedodatnią). Widać duże podobieństwo między tymi dwoma schematami blokowymi.

W obliczeniach komputerowych przyjmuje się najczęściej, że suma elementów zbioru, który jest pusty, wynosi zero.

Zmienna n , występująca w obu schematach, oznacza liczbę elementów w zbiorze — w algorytmie o schemacie 5.2a ta liczba jest wczytywana, a w algorytmie o schemacie 5.2b — jej wartość jest obliczana. Zmienna s oznacza sumę wczytywanych liczb. Zauważmy, że $s = 0$, gdy zbiór jest pusty i w tym przypadku dodatkowo przyjmujemy, że również średnia wynosi zero. ■

Ćwiczenie 5.4. W wybranym przez siebie języku programowania, Pascal lub Python, lub w obu, zapisz algorytmy obliczania średniej zbioru liczb dodatnich przedstawione na rysunku 5.2. w postaci schematów blokowych. Dobierz odpowiednie struktury danych. ■



Rysunek 5.2. Schematy blokowe algorytmów obliczania średniej ze zbioru liczb dodatnich: a) zbiór danych jest poprzedzony ich liczbą; b) zbiór danych jest zakończony wartownikiem — liczbą ujemną

5.3. Znajdowanie największego elementu

Zajmiemy się teraz jednym z najprostszych zadań — mając dany zbiór elementów, chcemy znaleźć w nim element największy (maksimum) lub najmniejszy (minimum). Dalej będziemy mówić tylko o sposobach wyznaczania maksimum, gdyż można je niemal automatycznie zastosować do znajdowania minimum. Największy i najmniejszy może oznaczać, w zależności od natury i rodzaju elementów: najwyższy i najniższy, najlepszy i najgorszy, zwycięzca i przegrany itd.

Jeśli ktoś zapyta nas, w jaki sposób wśród pewnych elementów znajdujemy największy, to udzielenie odpowiedzi na to pytanie może nas wprowadzić w zakłopotanie. Po pierwsze samo zadanie jest tak proste, że uznajemy takie pytanie niemal za żart. Po drugie po chwili namysłu zaczynamy jednak mieć wątpliwości, czy rzeczywiście potrafimy opisać sposób znajdowania największego elementu w zbiorze. A jeśli mielibyśmy opisać ten sposób jako algorytm, na przykład w postaci listy kroków?

Ćwiczenie 5.5. W rzeczywistych sytuacjach elementy największe i najmniejsze znajdujemy na ogół, posługując się metodami, które zależą od natury elementów. Na przykład chcesz znaleźć najwyższego i najniższego ucznia w swojej klasie. Sposób nasuwa się sam — prosisz wszystkich o powstanie, wtedy najwyższy uczeń zwykle jest natychmiast widoczny, chociaż z odszukaniem

najniższego będzie już kłopot, gdyż może on być zasłonięty przez wyższych uczniów (ta metoda nazywa się algorytmem *spaghetti* — czy potrafisz uzasadnić tę nazwę?). Nie możemy się spodziewać, że algorytm lub komputer będzie natychmiast „widział” te elementy w różnych zbiorach. Rozważ różne zbiory elementów i opisz powszechnie stosowane metody znajdowania w nich elementów największych i najmniejszych. Weź pod uwagę m.in. zbiory utworzone z 13 kart wybranych z talii 52 kart lub dat urodzenia wszystkich uczniów w klasie. Postaraj się zapisać te metody w postaci, która miałaby cechy algorytmu. ■

W tym punkcie wykluczamy rozwiązanie rozważanego problemu polegające na uporządkowaniu najpierw wszystkich elementów od największego do najmniejszego, a następnie wzięciu elementów z końców tego uporządkowania. W dalszej części rozdziału uzasadnimy, dlaczego teraz tak postępujemy.

Analizując nawet „naturalne metody” znajdowania największego i najmniejszego elementu, dostrzegamy w nich przeglądanie wszystkich elementów zbioru, bezpośrednio lub pośrednio. W przeciwnym bowiem razie element pominięty w rozważaniach mógłby się okazać szukany. Tak jest również wtedy, gdy chcesz wskazać najwyższego ucznia w klasie — swój wybór uzasadnisz tym, że jest to osoba wyższa od pozostałych. Wiedzie to nas do następującego algorytmu, w którym przyjmujemy, że zbiór danych jest złożony z liczb.

Algorytm Max — znajdowania największego elementu w zbiorze

Dane: Zbiór n liczb.

Wynik: Największy element max w danym zbiorze.

Krok 1. Przyjmij za max dowolny element ze zbioru.

Krok 2. Dla każdego innego elementu x zbioru wykonuj: jeśli x jest większe niż max , to za max przyjmij x . ■

Zauważ, że w kroku 2. tego algorytmu polecenie po dwukropku musi być wykonane dla wszystkich elementów zbioru, z wyjątkiem tego przyjętego w kroku 1. Nie będziemy jednak rozpisywać tego kroku bardziej szczegółowo.

Przez prostą modyfikację otrzymuje się Algorytm Min znajdowania min , najmniejszego elementu w zbiorze — spróbuj to zrobić.

Poniższe realizacje algorytmu Max w językach Pascal i Python nie powinny wyglądać tajemniczo nawet dla kogoś, kto nie zna tych języków — elementy zbioru danych są umieszczone w tablicy x o wymiarach od 1 do n lub liście lista. Ponieważ w tym przypadku jest znana liczba powtórzeń kroku 2., iterację można zrealizować za pomocą instrukcji `for`, w której jawnie występuje liczba przebiegów pętli.



```
function Max(n:integer; x: Tablica): integer;  
  var i,y:integer;  
begin  
  y := x[1];  
  for i := 2 to n do  
    if x[i] > y then y := x[i];  
  Max := y  
end; {Max}
```



```
def Max(lista):  
    Max = lista[0]  
    for i in range(1,len(lista),1):  
        if Max < lista[i]:  
            Max = lista[i]  
    return(Max)
```

Ćwiczenie 5.6. Zmodyfikuj opis algorytmu Max oraz funkcji Max tak, aby zamiast wartości największego elementu był znajdowany indeks tego elementu w przeszukiwanym ciągu. ■

Złożoność

Liczba porównań wykonywanych w algorytmie Max jest o jeden mniejsza od liczby elementów w zbiorze, czyli wynosi $n - 1$, gdy zbiór ma n elementów. Czy $n - 1$ porównań to dużo, by znaleźć największy element wśród n elementów? Byłoby za dużo, gdybyśmy potrafili to robić za pomocą mniejszej liczby tych działań. Przeformułujmy więc pytanie: czy potrafimy podać algorytm, który znajduje największy element w zbiorze za pomocą mniejszej liczby porównań, niż wykonuje to algorytm Max? Jest to już poważne wyzwanie — spróbujmy najpierw zasugerować inny algorytm.

Ćwiczenie 5.7. W algorytmie Max elementy zbioru danych są porównywane jeden po drugim. A gdyby jednocześnie wykorzystać wszystkie elementy: połączyć je w pary i porównać parami, tak jak jest rozgrywana pierwsza runda w turnieju tenisowym? Potem to samo powtórzyć dla zwycięzców, również parami. I tak dalej. Sprawdź, ile porównań wykonujesz w takim algorytmie, gdy $n = 8, 7, 11$ i 16 ? ■

Wykonanie tego ćwiczenia nie dostarcza jednak metody o mniejszej liczbie

porównań.

Ćwiczenie 5.8. Aby znaleźć największy element w zbiorze, trzeba wziąć pod uwagę wszystkie jego elementy. Najbardziej przekonującego argumentu co do takiego postępowania dostarcza nam sposób ustalania zwycięzców rozgrywek sportowych. Przypuśćmy, że turniej tenisowy jest rozgrywany systemem pucharowym, tzn. przegrywający odpada z gry (w meczach tenisowych nie ma remisów). Zastanów się, kiedy można powiedzieć, że dany zawodnik jest zwycięzcą turnieju? ■

Przytoczone w ćwiczeniu 5.8 uzasadnienie nosi nazwę **argumentu teorio-informacyjnego**. Ogólnie można powiedzieć, że jeśli wynik zależy od wszystkich danych, to każda z nich musi wystąpić w obliczeniach przynajmniej jeden raz. Ścisły dowód optymalności algorytmu Max znajduje się w podręczniku [EI-I]. Jest ciekawe, że chociaż od niepamiętnych czasów człowiek dokonywał najlepszych wyborów w podobny sposób, to dopiero w 1972 roku Ira Pohl po raz pierwszy podał formalny dowód optymalności tej naturalnej metody.

Odpowiedź w tym ćwiczeniu jest bardzo prosta — zwycięzcą turnieju jest zawodnik, który nie przegrał żadnego meczu, a każdy inny zawodnik przegrał przynajmniej jeden mecz. Licząc więc tylko przegrane w turnieju mecze, należy ich rozegrać przynajmniej $n - 1$, gdzie n jest liczbą zawodników. Jest to przepiękny argument Hugona Steinhausa z jego *Kalejdoskopu matematycznego* [Steinhaus].

Z tych rozważań wynika, że w każdym algorytmie, w którym porównuje się elementy danych, aby znaleźć wśród nich największy, należy wykonać przynajmniej $n - 1$ porównań, gdzie n jest liczbą elementów w zbiorze. Algorytm Max jest zatem **optymalny** pod względem złożoności obliczeniowej, gdyż wykonuje najmniejszą możliwą liczbę porównań. Algorytm ten nie jest jednak najlepszą propozycją rozegrania turnieju tenisowego, zwłaszcza dla zwycięzcy, który musiałby rozegrać aż $n - 1$ meczów — wrócimy do tej kwestii w następnym punkcie. Turnieje są zwykle rozgrywane metodą opisaną w ćwiczeniu 5.7.

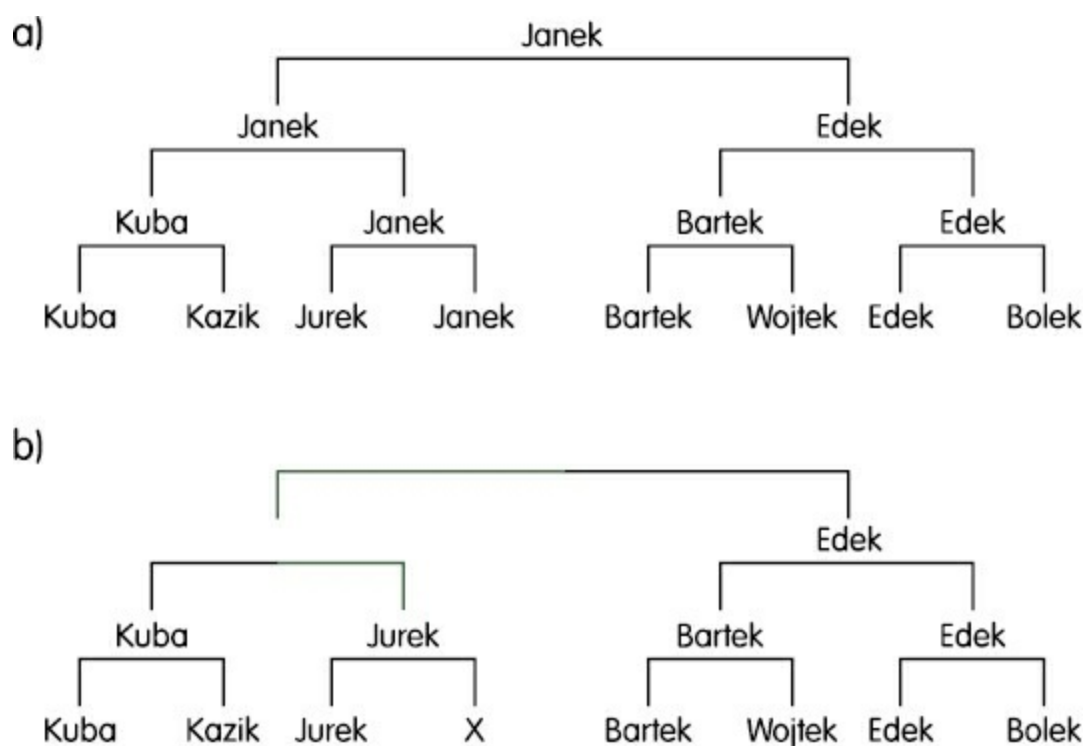
5.4. Kompletowanie podium zwycięzców

Po raz pierwszy w tej książce pojawia się obok nazwisko Hugona Steinhausa, polskiego matematyka, twórcy wielu kierunków badań w matematyce, popularyzatora matematyki i jej zastosowań. W swej znakomitej, ponadczasowej książce pt. *Kalejdoskop matematyczny* zawarł on wiele pięknych rozwiązań problemów matematycznych, które są prześiąknięte głębokimi ideami algorytmicznymi.

Około 1930 roku Hugo Steinhaus zastanawiał się, 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 pokonany w finale jest rzeczywiście drugim najlepszym zawodnikiem turnieju, a co za tym idzie — jest lepszy od wszystkich pozostałych zawodników.

Aby się przekonać, że wątpliwości H. Steinhausa były uzasadnione, spójrzmy na drzewo turnieju przedstawione na rysunku 5.3a. 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 (który 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? Odpowiedział na to inny polski matematyk, Józef Schreier w 1932 roku.



Rysunek 5.3. Drzewo: a) przykładowych rozgrywek w turnieju tenisowym; b) znajdowania drugiego najlepszego zawodnika turnieju

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 byłiby wyłonieni w $2n - 3$ meczach, $(n - 1) + (n - 2) = 2n - 3$. Hugo Steinhaus oczywiście znał to rozwiązanie, pytał więc o najmniejszą potrzebną liczbę meczów, i takiej

odpowiedzi udzielił J. Schreier.

Jeśli chcemy, aby drugi najlepszy zawodnik nie musiał być wyłaniany w nowym pełnym turnieju, to musimy umieć skorzystać z wyników głównego turnieju. Posłużymy się drzewem turnieju z rysunku 5.3a. Zauważmy, że Edek jest oczywiście najlepszy wśród zawodników, którzy w drzewie rozgrywek znajdują się w wierzchołkach leżących poniżej wierzchołka, który on zajmuje. Musimy więc jedynie porównać go z zawodnikami drugiego poddrzewa. Aby i w tym poddrzewie wykorzystać wyniki dotychczasowych meczów, musimy wyeliminować z niego Janka — zwycięzcę turnieju i wstawić na jego miejsce drugiego najlepszego zawodnika w tym poddrzewie. Najprościej to zrobić, przyjmując że miejsce Janka w wierzchołku końcowym zajmuje jakiś zawodnik X, słabszy od wszystkich pozostałych zawodników turnieju. (Jeśli rozważalibyśmy porządkowanie liczb, a nie tenisistów, to w miejsce Janka należałoby przyjąć jakąś bardzo małą liczbę, mniejszą od wszystkich liczb w rozważanym zbiorze). Spowoduje to, że zawodnik X przegra pierwszy mecz — wygra więc Jurek i dalej trzeba powtórzyć tylko tę część turnieju, która znajduje się na drodze od wierzchołka, w którym znajduje się teraz Jurek, do korzenia — na rysunku 5.3b oznaczyliśmy ją zielonym kolorem. Aby wyłonić drugiego najlepszego zawodnika turnieju, trzeba w tym przypadku rozegrać dwa dodatkowe mecze: Kuba przeciwko Jurkowi i zwycięzca tego meczu przeciwko Edkowi. Innym sposobem wyłonienia drugiego zawodnika tego turnieju jest umieszczenie Edka w miejscu X i rozegranie części turnieju w lewym poddrzewie.

Algorytm ten, po zmianie słownictwa, można zastosować do znajdowania największej i drugiej największej liczby w zbiorze danych (zobacz zadanie 5.1).

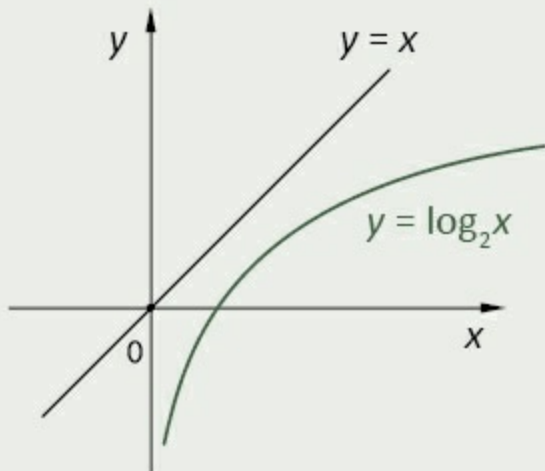
Po raz pierwszy w naszych rozważaniach o algorytmach pojawia się funkcja logarytmiczna, która odgrywa bardzo ważną rolę w algorytmice, zwłaszcza w analizie złożoności obliczeniowej algorytmów, patrz [Log]. **Funkcja logarytmiczna** jest funkcją odwrotną do funkcji potęgowej w następującym sensie: jeśli $n = 2^k$, to wartością $\log_2 n$ jest k . Funkcja $\log_2 n$ rośnie o wiele wolniej niż funkcja liniowa n . Na rysunku 5.4 przedstawiamy wykresy tych funkcji dla zmiennej rzeczywistej x oraz tabelę z ich wartościami dla kilku liczb naturalnych n .

Złożoność

T

Ile porównań wykonujemy w opisanym algorytmie? Najlepszy zawodnik jest wyłaniany w $n - 1$ meczach, gdzie n jest liczbą wszystkich zawodników — o tym była mowa w poprzednim punkcie. Z kolei aby wyłonić drugiego najlepszego zawodnika, trzeba rozegrać tyle meczów, ile jest poziomów w drzewie turnieju głównego (z wyjątkiem jednego 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 $\log_2 n$. Liczba porównań wynosi więc $(n - 1) + (\log_2 n - 1) = n + \log_2 n - 2$. 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. ■



n	$\log_2 n$
16	4
128	7
1024	10
131072	17
1048576	20
16777216	24

Rysunek 5.4. Wykresy funkcji liniowej i logarytmicznej oraz tabela z wartościami funkcji logarytmicznej dla kilku wybranych liczb naturalnych.

Ćwiczenie 5.9. Sporządź tabelę podobną do zamieszczonej na rysunku 5.4, ale zawierającą wartości funkcji $2n - 3$ oraz $n + \log_2 n - 2$, odpowiadające liczbie porównań wykonywanych w dwóch omówionych wyżej algorytmach znajdowania najlepszego i drugiego najlepszego zawodnika turnieju. Dla ułatwienia, obliczenia wykonaj dla liczb n będących potęgami liczby 2. ■

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 5.10. 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, to 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** i może być stosowana również do porządkowania liczb (zobacz problem 5.3), [Knuth-3].

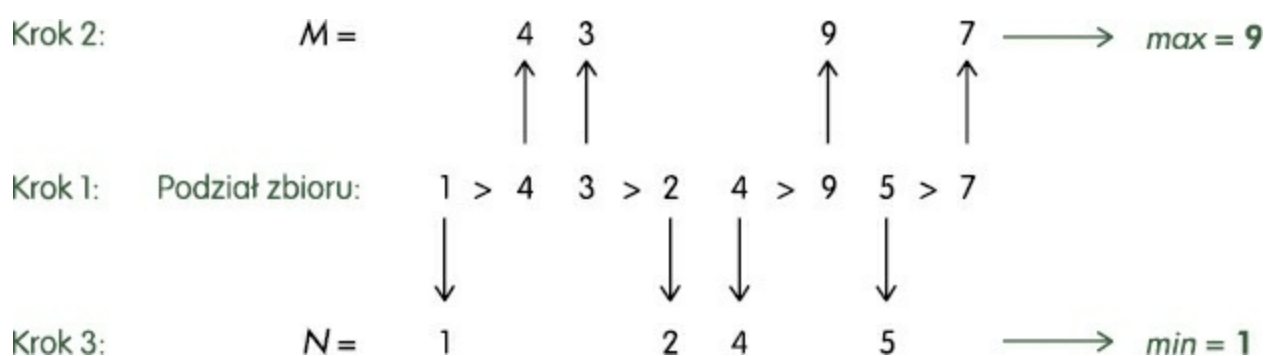
5.5. Znajdowanie jednocześnie największego i najmniejszego elementu

Aby znaleźć **rozpiętość zbioru** n liczb, wystarczy odszukać wśród nich maksimum i minimum, a potem obliczyć ich różnicę. Możemy w tym celu zastosować poznane już algorytmy Max i Min. W każdym z nich liczba porównań wynosi $n - 1$, a więc rozpiętość zbioru n liczb można obliczyć, wykonując $2n - 2$ porównania. Tę liczbę działań można jeszcze zmniejszyć o jedno porównanie, usuwając maksimum ze zbioru danych przed wyznaczaniem w nim minimum. Czy to jest jedyna oszczędność, jaką możemy zrobić?

Wyprowadzimy teraz algorytm znajdowania maksimum i minimum, w którym wykonujemy znacznie mniej porównań i ponadto znajdujemy te dwie liczby jednocześnie. Zauważmy, że ta wymyślona *ad hoc* metoda z poprzedniego akapitu nie jest w gruncie rzeczy jednoczesnym wyznaczaniem obu tych wielkości. Najpierw bowiem wyznaczamy maksimum, a później minimum.

W algorytmach Max i Min z każdego wykonanego sprawdzenia, czy $x > y$, wyciągamy jeden wniosek: x jest kandydatem na maksimum — w algorytmie Max, natomiast y jest kandydatem na minimum — w algorytmie Min. Niestety, wykonując algorytmy Max i Min niezależnie, jeden po drugim, tę nierówność sprawdzamy dwa razy, aby uzyskać oba te wnioski. Spróbujmy połączyć działanie obu algorytmów, przynajmniej na początku. Porównajmy najpierw kolejne pary elementów — takich operacji wykonamy około $n/2$. Z każdego porównania, mniejsza liczba jest kandydatem na minimum, a większa — kandydatem na maksimum. Zatem w drugim etapie będziemy szukać maksimum i minimum w zbiorach już o połowę mniejszych. Opiszemy teraz ten algorytm w

postaci listy kroków. Jego działanie jest zilustrowane na rysunku 5.5.



Rysunek 5.5. Przykład działania algorytmu MaxMin

Algorytm MaxMin — znajdowania w zbiorze maksimum i minimum jednocześnie

Dane: Zbiór n liczb.

Wyniki: \max i \min , odpowiednio największy i najmniejszy element w zbiorze danych.

Krok 1. {Podział zbioru danych na dwa podzbiory.} Połącz elementy zbioru danych w pary. Jeśli n jest liczbą nieparzystą, to jeden z elementów pozostaje wolny — oznaczmy go przez z . Porównaj elementy w parach: przypuśćmy, że dla pary x i y mamy $x > y$; wtedy dołącz x do zbioru kandydatów na maksimum — oznaczmy ten zbiór przez M , a y do zbioru kandydatów na minimum — oznaczmy go przez N .

Krok 2. Znajdź \max w zbiorze M za pomocą algorytmu Max.

Krok 3. Znajdź \min w zbiorze N za pomocą algorytmu Min.

Krok 4. Jeśli n jest liczbą nieparzystą, to: jeśli $z < \min$, to za \min przyjmij z , jeśli $z > \max$, to za \max przyjmij z . ■

To nie jest pomyłka w nazwie tej zasady — nie należy jej mylić ze starożytną zasadą dziel i rządź (łac. *divide et impera*), odnoszącą się rzeczywiście do władania nad rozległymi terytoriami Cesarstwa Rzymskiego. Zasada, o której tutaj mówimy, pochodzi od angielskiej nazwy *divide and conquer*. Spróbuj doszukać się różnicy w tych zasadach na podstawie ich nazw.

Algorytm MaxMin ma wiele ciekawych własności. Przede wszystkim jest bardzo prostym przykładem postępowania zgodnie z **zasadą dziel i zwyciężaj**.

Pierwsza część tej zasady jest zilustrowana na rysunku 5.5. — w kroku 1. tego algorytmu zbiór danych jest dzielony na dwa rozłączne podzbiory, prawie równoliczne, dla których w dwóch następnych krokach są rozwiązywane podobne problemy. Zatem problem MaxMin rozpada się na dwa problemy, Max i Min, na rozłącznych podzbiorach o takiej samej liczbie elementów.

Uzasadnienie drugiej części tej zasady pojawi się trochę dalej, gdy będziemy

dokładnie omawiać złożoność tego algorytmu.

Przed omówieniem realizacji algorytmu MaxMin wykonaj następane ćwiczenie, by się przekonać, jak on działa, gdy liczba elementów w zbiorze jest nieparzysta.

Ćwiczenie 5.11. Zastosuj algorytm MaxMin dla zbioru pierwszych siedmiu liczb z przykładu pokazanego na rysunku 5.5. Ile wykonałeś porównań? ■

Kilka ogólnych uwag dotyczących realizacji algorytmu MaxMin, które nie zależą od sposobu jego przedstawienia. Przypuśćmy, że zbiór danych jest zapisany w tablicy. W algorytmie MaxMin ten zbiór jest dzielony na dwa podzbiory M i N . Aby uniknąć tworzenia tych podzbiorów w nowym miejscu, można zastosować prostsze rozwiązanie: przyjmijmy, że jeśli w wyniku porównania dwóch kolejnych liczb x i y otrzymamy $x > y$, to będziemy te elementy przestawiać. W ten sposób po pierwszym kroku algorytmu na nieparzystych miejscach na taśmie znajdą się elementy zbioru N , a na parzystych — elementy zbioru M . Ponadto można ujednolicić kroki 2. i 3. oraz pozbyć się kroku 4.: jeśli liczba n jest nieparzysta, to przedłużymy ciąg, powielając ostatni jego element. Dzięki temu najmniejszy element zbioru będzie szukany w połowie elementów w tablicy o indeksach nieparzystych, a największy — w drugiej połowie, o indeksach parzystych. Przy takiej organizacji danych, dla przykładu przedstawionego na rysunku 5.5, po wykonaniu kroku 1. algorytmu liczby w ciągu danych będą występowały w kolejności: 1, 4, 2, 3, 4, 9, 5, 7, a dla zbioru złożonego z pierwszych siedmiu liczb — w kolejności: 1, 4, 2, 3, 4, 9, 5, 5.

Ćwiczenie 5.12. Wykonaj algorytm MaxMin z modyfikacją opisaną w poprzednim akapicie dla zbiorów danych z rysunku 5.5 oraz z ćwiczenia 5.11. Ile porównań wykonałeś w obu przypadkach? Porównaj działanie obu wersji algorytmu dla danych zaczerpniętych z poprzedniego ćwiczenia. ■

Algorytm MaxMin ma jeszcze jedną bardzo korzystną własność: kroki 2. i 3. odwołują się do znanych algorytmów znajdowania w zbiorze elementu największego i najmniejszego. Algorytm, który występuje jako część innego algorytmu, nazywa się **podprogramem** lub **procedurą**, tak jak w języku Pascal (dalej określeń podprogram i procedura będziemy używać zamiennie). Zatem w realizacji algorytmu MaxMin możemy wprost skorzystać z podprogramów wykonujących te dwa kroki. Na ogół musimy jednak najpierw dostosować opis podprogramu do struktury danych w algorytmie głównym. W tym przypadku elementy największy i najmniejszy są szukane w krokach 2. i 3. w ciągach złożonych z co drugiego elementu tablicy, a algorytm Max opisany w punkcie 5.3 działa na danych, które są kolejnymi elementami tablicy. Są to jednak niewielkie modyfikacje realizacji algorytmu Max. Gdy dysponujemy realizacją algorytmu Max, zbudowanie realizacji algorytmu Min wymaga tylko kilku ingerencji.

Jeszcze wielokrotnie w tej książce w rozwiązaniu jednego problemu będziemy korzystali z rozwiązań innych problemów — jest to jedna z podstawowych zasad i technik stosowanych przy rozwiązywaniu problemów.

Poniżej zamieszczamy realizację algorytmu MaxMin w postaci procedury w języku Pascal, a wykonanie opisu w języku Python pozostawiamy Czytelnikowi.



```
procedure MaxMin(n:integer;x:Tablica1n;var Max,Min:integer);  
  var i,y:integer;  
begin  
  i := 2;  
  while i <= n do begin {Podział ciagu}  
    if x[i-1] > x[i] then begin  
      y := x[i-1]; x[i-1] := x[i]; x[i] := y  
    end;  
    i := i + 2  
  end;  
  Min := x[1];  
  if n = 1 then Max := x[1]  
  else begin  
    Max := x[2]; i := 4;  
    while i <= n do begin  
      if Min > x[i-1] then Min := x[i-1];  
      if Max < x[i] then Max := x[i];  
      i := i + 2  
    end  
  end; {n > 1}  
  {Gdy n jest nieparzyste i n > 1}  
  if (n > 1) and (m = i - 1) then begin  
    if Min > x[n] then Min := x[n];  
    if Max < x[n] then Max := x[n]  
  end  
end; {MaxMin}
```

Złożoność

Obliczymy teraz, ile porównań wykonujemy w algorytmie MaxMin. Dla uproszczenia przyjmijmy, że n jest liczbą parzystą. W kroku 1. wykonujemy $n/2$ porównań, które powodują podzielenie zbioru na dwa podzbiory. Z kolei w

krokach 2. i 3. jest wykonywanych $n/2 - 1$ takich działań w każdym. W sumie liczba porównań algorytmu MaxMin wynosi $3n/2 - 2$, gdy n jest liczbą parzystą. Natomiast gdy n jest liczbą nieparzystą, liczbę porównań algorytmu zapisujemy w postaci $\lceil 3n/2 \rceil - 2$, posługując się funkcją powała (definicja tej funkcji jest podana w ramce obok problemu 5.4), zobacz zadanie 5.2.

Przedstawiony algorytm jednoczesnego znajdowania największego i najmniejszego elementu w zbiorze jest również algorytmem optymalnym. Dowód tego stwierdzenia jest jednak trudniejszy niż w przypadku algorytmu znajdowania tylko jednej z tych wielkości.

Porównując złożoność algorytmu MaxMin ze złożonością metody *ad hoc*, możemy dostrzec „zwycięstwo” tej pierwszej nad tą drugą. To służy za uzasadnienie drugiego członu **zasady**, którą posłużyliśmy się przy wyprowadzeniu tego algorytmu — **dziel i zwyciężaj**. Informacja w ramce powyżej świadczy dodatkowo, że otrzymaliśmy najlepszy z możliwych algorytmów.

Do problemu jednoczesnego znajdowania maksimum i minimum wrócimy w punkcie 9.1, przedstawiając metodę rekurencyjną opartą na tej samej zasadzie.

5.6. Obliczanie innych miar centralności danych

Na początku tego rozdziału wprowadziliśmy pojęcia, które określają skupienie lub centralność danych. Te miary stosujemy wtedy, gdy chcemy ocenić, na ile pewne tendencje w danych są zbieżne, a więc, czy temperatura utrzymuje się blisko jakiejś wartości, czy uczniowie w klasie są podobnego wzrostu, czy podoba nam się ten sam rodzaj muzyki itd. W punkcie 5.2 omówiliśmy obliczanie średniej ze zbioru liczb, w punkcie 5.5 przedstawiliśmy szybki sposób znajdowania najbardziej krańcowych danych, których wartości można wykorzystać np. do obliczenia rozstępu danych. Pozostały do omówienia sposoby wyznaczania mediany i mody, czyli miar centralności danych.

Zwróćmy tutaj uwagę, że do wyznaczania największego i najmniejszego elementu w zbiorze oraz drugiego i trzeciego największego elementu zbudowaliśmy specjalne algorytmy, chociaż najprościej byłoby uporządkować najpierw wszystkie elementy, na przykład od najmniejszego, i wtedy sięgnąć po odpowiednie z nich: najmniejszy jest na początku, największy na końcu, drugi największy — bezpośrednio przed nim. Jednak postąpiliśmy tak przynajmniej z dwóch względów. Po pierwsze pokazaliśmy, że do znalezienia największej i najmniejszej liczby w ciągu wcale nie trzeba znać porządku wszystkich liczb — utworzone przez nas algorytmy są o wiele prostsze niż porządkowanie całego ciągu. Po drugie dzięki temu, że są one prostsze, wykonują mniej działań. (W

punkcie 10.4 uzasadniamy, że najlepsza metoda porządkowania ciągu n elementów polega na wykonaniu co najmniej $n \log_2 n$ porównań, a jest to funkcja o wartościach o wiele większych niż funkcje $n - 1$, $3n/2 - 3$, $2n + 1$ występujące jako złożoności w tym rozdziale).

Nie potrafimy niestety zbudować prostych algorytmów znajdowania mediany i mody. W przypadku tych dwóch wielkości rzeczywiście najprościej byłoby najpierw uporządkować dane — na przykład w porządku niemalejącym. Wtedy medianę i modę można łatwo znaleźć, korzystając z tego uporządkowania.

Mediana zbioru jest środkowym elementem tego zbioru w jego uporządkowaniu. Jeśli zbiór zawiera nieparzystą liczbę elementów, to ma jeden środkowy element i jest on medianą tego zbioru. Jeśli natomiast zawiera parzystą liczbę elementów, to za medianę przyjmuje się na ogół jeden z dwóch środkowych elementów.



W metodzie znajdowania mediany na podstawie ciągu uporządkowanego najwięcej czasu zabiera jego porządkowanie. Istnieje szybsza metoda znajdowania mediany, która nie porządkuje ciągu — jest ona opisana w książce [BanKrecz]. Ta metoda polega na skorzystaniu z algorytmu, który znajduje k -ty co do wielkości element w zbiorze, i przyjęciu za medianę elementu znajdującego się na pozycji $\lceil n/2 \rceil$, gdzie n jest liczbą elementów w zbiorze. ■

Moda jest najczęściej występującym elementem w zbiorze, przy tym zakładamy, że jeśli każdy element występuje w zbiorze dokładnie jeden raz, to zbiór nie ma mody. Jeśli zbiór jest uporządkowany, to takie same elementy występują w nim obok siebie w jednym odcinku tego uporządkowania. Wystarczy więc przejrzeć elementy zbioru zgodnie z ich uporządkowaniem, zliczać, ile razy się powtarza każdy element, i pamiętać ten, który powtarza się najczęściej.

Ćwiczenie 5.13. Zapisz, w wybranej przez siebie reprezentacji, algorytm znajdowania mody w uporządkowanym zbiorze. Przyjmij, że zbiór jest dany w tablicy lub w pliku. ■

Modę można łatwo „dostrzec”, analizując wykres zbioru, zwany jego histogramem. **Histogram** jest słupkowym wykresem częstości występowania elementów zbioru. Wykresy te łatwo otrzymuje się w arkuszu kalkulacyjnym.

5.6.1. Znajdowanie lidera w zbiorze

Szczególnym przypadkiem problemu znajdowania mody zbioru jest sprawdzenie,

czy zbiór zawiera lidera i ewentualne znalezienie go. **Liderem** nazywamy element, który występuje w zbiorze więcej niż połowę razy, czyli więcej niż $n/2$ razy, gdzie n jest liczbą elementów zbioru. Na przykład zbiór $\{1, 2, 1, 3, 1, 4\}$ nie zawiera lidera, gdyż liczba 1 występuje w nim dokładnie $n/2 = 6/2 = 3$ razy, ale zbiory $\{1, 2, 1, 3, 1, 1\}$ i $\{1, 2, 1, 3, 1, 4, 1\}$ zawierają lidera — jest nim liczba 1. W ostatnim zbiorze częstość występowania lidera jest równa 4, natomiast $n/2 = 7/2 = 3.5$.

Problem znajdowania lidera w zbiorze ma zastosowanie na przykład przy liczeniu głosów w wyborach, w których zwycięzca musi uzyskać więcej niż połowę głosów. Elementami zbioru są w tym przypadku nazwiska (lub numery) kandydatów. Jeśli w wyborach startuje kilku kandydatów, dwóch lub trzech, to najłatwiej zliczyć głosy, odkładając kartki z głosowania na stosy przyporządkowane kandydatom (zobacz punkt 6.3.1, gdzie przedstawiamy algorytm, w którym te kartki są odkładane do kubeków). W dzisiejszych wyborach na ogół startuje wielu kandydatów i dobrze jest dysponować jeszcze inną metodą, być może szybszą.

Ćwiczenie 5.14. Postaraj się, najpierw na przykładach, znaleźć relacje między trzema elementami zbioru: medianą, modą i liderem. ■

Na podstawie wyników ostatniego ćwiczenia znajdowanie lidera w zbiorze można zastąpić określeniem innych miar centralności. Pierwsza metoda szybko narzuca się sama: znajdź modę i sprawdź, czy jej częstość w zbiorze jest większa niż wartość $n/2$. By znaleźć modę, musimy jednak najpierw uporządkować zbiór, a ta operacja jest pracochłonna, zwłaszcza na dużym zbiorze. Prostszy jest algorytm polegający na znalezieniu mediany, czyli elementu znajdującego się na pozycji $n/2$, i sprawdzeniu, czy jest on liderem (bowiem jeśli zbiór ma lidera, to jest on jego medianą — czy dostrzegłeś to, rozwiązując ćwiczenie 5.14?). Wiemy już, że istnieje algorytm znajdowania mediany, w którym nie korzysta się z uporządkowania zbioru, ale jego opis jest złożony [BanKrecz].

T

Omówimy^[3] teraz bardzo „elegancki” algorytm znajdowania lidera w zbiorze. W tej metodzie wykorzystuje się dość proste, ale nieco zaskakujące spostrzeżenie: Jeśli zbiór X zawiera lidera, a x i y są dwoma jego **różnymi** elementami, to zbiór $Y = X \setminus \{x, y\}$, który powstał przez usunięcie elementów x i y ze zbioru X , również zawiera lidera.

Zwróćmy jednak od razu uwagę, że odwrotne stwierdzenie nie jest prawdziwe — to znaczy zredukowany zbiór Y może zawierać lidera, chociaż w zbiorze X może go nie być. Na przykład zbiór $X = \{1, 2, 2, 3, 4\}$ nie zawiera lidera, ale po

usunięciu z niego dwóch różnych elementów, np. liczb 1 i 3, pozostały zbiór $\{2, 2, 4\}$ zawiera lidera, którym jest liczba 2; jednak nie jest ona liderem w naszym początkowym zbiorze X .

Uzasadnienie słuszności tego spostrzeżenia jest proste. Oznaczmy przez l element zbioru X , który jest jego liderem. Jeśli x i y są dwoma różnymi elementami zbioru X , to co najwyżej jeden z nich jest równy l , zatem w zredukowanym zbiorze Y (w porównaniu ze zbiorem X) liczba elementów jest mniejsza o dwa, a częstość występowania w zbiorze elementu l jest mniejsza o co najwyżej jeden, czyli l pozostaje liderem w zbiorze Y . Musimy jedynie sprawdzić, czy lider zbioru Y jest liderem początkowego zbioru X — w algorytmie podanym niżej robimy to na końcu.

Algorytm ten składa się z dwóch etapów. W pierwszym (kroki 2. – 5.) szukamy kandydata na lidera, a w drugim — sprawdzamy, czy znaleziony kandydat jest rzeczywiście liderem zbioru. Oznaczamy przez l kandydata na lidera, a przez c — jego krotność określoną na podstawie kolejnych porównań.

Algorytm znajdowania lidera w zbiorze

Dane: Zbiór X złożony z n elementów $\{x_1, x_2, \dots, x_n\}$.

Wynik: Lider l w zbiorze X lub informacja, że X nie zawiera lidera.

Krok 1. Przyjmij x_1 za lidera l oraz niech $c := 1$.

{Kroki 2. – 5. stanowią etap wykrywania kandydata na lidera w zbiorze X .}

Krok 2. Dla kolejnych wartości $i = 2, 3, \dots, n$ wykonaj kroki 3. – 5., a następnie przejdź do kroku 6.

Krok 3. Jeśli $c = 0$, to wykonaj krok 4., a w przeciwnym razie wykonaj krok 5.

Krok 4. {Obieramy nowego kandydata na lidera.} Przyjmij x_i za lidera l i niech $c := 1$.

Krok 5. {Porównujemy kolejny element zbioru X z kandydatem na lidera.} Jeśli $x_i = l$, to zwiększ c o 1, a w przeciwnym razie zmniejsz c o 1. {W pierwszym przypadku, kolejny element jest równy liderowi, a w drugim — nie jest równy, usuwamy go więc wraz z jedną z krotności lidera.}

{W krokach 6. – 8. sprawdzamy, czy l jest rzeczywiście liderem w zbiorze X .}

Krok 6. Jeśli $c = 0$, to przejdź do kroku 7., w przeciwnym razie przejdź do kroku 8.

Krok 7. Zbiór X nie ma lidera. Zakończ algorytm.

Krok 8. Wyznacz, ile razy kandydat na lidera l występuje w zbiorze X . Jeśli ta

liczba jest większa od $n/2$, to l jest rzeczywiście liderem w zbiorze X , w przeciwnym razie ten zbiór nie zawiera lidera. ■

Poprawność tego algorytmu wynika z dyskusji poprzedzającej opis. Dodatkowe wyjaśnienia i uzasadnienia zawarliśmy w komentarzach.

Obliczymy teraz, ile porównań wykonujemy w algorytmie. W pierwszym etapie dla każdego elementu zbioru X (z wyjątkiem pierwszego) jest wykonywane jedno takie działanie w kroku 3. lub 5. W etapie drugim porównania są wykonywane w kroku 6. (jedno) i w kroku 8. ($n + 1 - n$ porównań, by obliczyć wielokrotność l w zbiorze X i jedno porównanie, by sprawdzić, czy ta wielokrotność jest większa od $n/2$). W sumie liczba porównań wynosi $(n - 1) + (n + 2) = 2n + 1$.

Otrzymaliśmy więc algorytm znajdowania w zbiorze lidera, który jest szybszy od wcześniej wspomnianych metod, w których wykorzystuje się porządkowanie i znajdowanie mediany lub mody w zbiorze X . Algorytm ten jest też o wiele prostszy koncepcyjnie i może dlatego wykonuje tak niewiele działań. Równie łatwo można znajdować idola w towarzystwie — zobacz problem 13.22.

5.7. Zadania i problemy

Problem 5.1. Inną statystyką rozważaną jako miara centralności danych jest **odchylenie standardowe**, definiowane następująco:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}},$$

gdzie n jest liczbą danych w ciągu x_1, x_2, \dots, x_n , a \bar{x} jest średnią tych liczb.

Odchylenie standardowe może świadczyć o rozrzucie elementów wokół średniej — jest tym mniejsze, im dane są bliżej skupione wokół średniej. W punkcie 5.2 opisaliśmy dwa algorytmy obliczania wartości średniej, które mają tę cechę, że wyznaczamy w nich średnią, przeglądając jednokrotnie ciąg danych. Powyższy wzór sugeruje, że chcąc obliczyć odchylenie standardowe, najpierw powinniśmy obliczyć średnią, a dopiero później składniki sumy znajdującej się w liczniku ułamka pod pierwiastkiem. Spróbuj przekształcić ten wzór tak, aby odchylenie standardowe ciągu liczb można było obliczać, przeglądając ciąg danych jednokrotnie. Korzystając z przekształconego wzoru, rozbuduj wybrany algorytm obliczania średniej o dodatkowe obliczanie odchylenia standardowego.

Problem 5.2. Przeszukiwanie nieuporządkowanego ciągu (zobacz punkt 5.1.2) w poszukiwaniu danego elementu może się kończyć przejrzaniem całego ciągu.

Założ teraz, że masz nadal stosować przeszukiwanie liniowe, ale ciąg, w którym poszukujesz danego elementu, został wcześniej uporządkowany. Podaj opis algorytmu liniowego przeszukiwania uporządkowanego ciągu. Jeśli dany element nie znajduje się w ciągu, to algorytm powinien dodatkowo określać takie w nim miejsce, że po umieszczeniu w nim tego elementu ciąg pozostaje uporządkowany. Taką metodę nazywa się **algorytmem liniowego umieszczania** w ciągu uporządkowanym. Porównaj efektywność przeszukiwania liniowego ciągu nieuporządkowanego z efektywnością takiego przeszukiwania ciągu uporządkowanego.

Uwaga. Algorytm liniowego umieszczania jest wykorzystany w algorytmie porządkowania przez umieszczanie, zobacz punkt 10.1.

Zadanie 5.1. Wzorując się na metodzie znajdowania najlepszego i drugiego najlepszego zawodnika turnieju tenisowego, podaj w postaci listy kroków opis algorytmu znajdowania największej i drugiej największej liczby w zbiorze. Utwórz realizację tego algorytmu w postaci programu.

Problem 5.3. Omówiony w punkcie 5.4 sposób wyłaniania trzech rzeczywiście najlepszych zawodników turnieju tenisowego może być rozszerzony na znajdowanie uszeregowania wszystkich zawodników turnieju — ten algorytm nazywamy metodą **porządkowania na drzewie**. Podaj opis tego algorytmu w postaci listy kroków i zastosuj go do uporządkowania wybranego zbioru liczb.

Problem 5.4. Narysuj wykresy funkcji podłoga x i powała x — definicje tych funkcji podajemy w ramce poniżej.

Funkcje **podłoga** $x(\lfloor x \rfloor)$ i **powała** $x(\lceil x \rceil)$ przyporządkowują zmiennej

rzeczywistej x najbliższe jej wartości liczby całkowitej. Podłoga x jest największą liczbą całkowitą nie większą niż x , a powała x — jest najmniejszą liczbą całkowitą nie mniejszą niż x . Na przykład:

$$\lfloor -3.5 \rfloor = -4, \lceil -3.5 \rceil = -3, \lfloor 1.7 \rfloor = 1, \lceil 1.7 \rceil = 2, \lfloor 2 \rfloor = \lceil 2 \rceil = 2.$$

Zadanie 5.2. Sprawdź, że gdy n jest liczbą nieparzystą, wówczas algorytm MaxMin wykonuje $\lceil 3n/2 \rceil - 2$ porównania.

Zadanie 5.3. Poznałeś algorytmy znajdowania podstawowych wielkości charakteryzujących zbiór danych: maksimum i minimum, rozpiętość, średnią, medianę, modę i lidera. Przypuśćmy, że dla pewnego zbioru danych obliczyłeś wartości tych wielkości i nagle okazało się, że pominąłeś w swoich rachunkach jedną daną. Zaproponuj, w jaki sposób poprawić wartości tych wielkości, uwzględniając pominiętą daną. Dla każdej z wielkości opisz algorytm, w którym będą wykonywane odpowiednie obliczenia korygujące.

Mogłeś dowiedzieć się, że:

- ▶ zbiór — w matematyce nieuporządkowany — w algorytmach występuje jako ciąg dowolnie uporządkowanych elementów, które dodatkowo mogą się powtarzać;
- ▶ rolę **wartownika w zbiorze** — elementu stojącego na krańcach zbioru — jest „pilnowanie”, by żadne obliczenia dotyczące zbioru nie wykroczyły poza jego elementy;
- ▶ nie ma lepszego (tj. szybszego) algorytmu znajdowania najmniejszego lub największego elementu w zbiorze niż powszechnie stosowany sposób;

oraz poznać:

- ▶ algorytm **przeszukiwania liniowego**, w którym po prostu porównuje się kolejno element po elemencie;
- ▶ sposób wyłaniania trzech rzeczywiście najlepszych zawodników w turnieju tenisowym;
- ▶ pierwszy przykład użycia strategii **dziel i zwyciężaj** do znajdowania jednocześnie największego i najmniejszego elementu w zbiorze — jest to również **algorytm optymalny**;
- ▶ pierwszy przykład projektu algorytmu złożonego z **podprogramów**;
- ▶ sposób wyłaniania **lidera w grupie**, czyli osoby wybranej przez więcej niż połowę członków grupy.

[1] W tej książce, odmiennie niż na lekcjach matematyki, pojęcia „zbiór” używamy na oznaczenie kolekcji elementów, wśród których niektóre z nich mogą się powtarzać — w matematyce takie zbiory nazywa się **multizbiorami**.

[2] Tablice w języku Pascal i listy w języku Python jako struktury danych różnią się operacjami, jakie można na nich wykonywać. W tej książce korzystamy z podstawowych własności tych struktur danych i operacji, które można na nich wykonywać, a po dokładne wyjaśnienia odsyłamy do podręczników programowania w tych językach.

[3] Dalsze fragmenty tego rozdziału nie są jednak zbyt trudne do śledzenia.

Rozdział 6. Porządkowanie ciągu elementów

Tu poznasz podstawowe **algorytmy porządkowania** zbiorów zawierających dowolną liczbę elementów:

- ▶ algorytm **bąbelkowy** wynoszenia największych elementów jak najwyżej;
- ▶ algorytm porządkowania **przez wybieranie** kolejnych co do wielkości elementów;
- ▶ algorytmy porządkowania **kubelkowego** i **pozycyjnego**, w których wykorzystuje się wartości porządkowanych elementów.

Początkowy fragment tego rozdziału jest wprowadzeniem do porządkowania jako działu algorytmiki i zawiera podstawowe informacje o tym problemie i jego znaczeniu. Następnie w punktach 6.1 – 6.3 przedstawiamy algorytmy porządkowania dowolnej liczby elementów, którymi mogą być liczby oraz elementy o bardziej złożonej postaci (takie jak słowa i daty).

Hugo Steinhaus w swoim *Kalejdoskopie matematycznym* używa jeszcze jednego określenia na porządkowanie — **szeregowanie**.

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 alfabety, słowa w encyklopedii, daty, kredki według kolorów czy nawet osoby według wzrostu), to wykonywanie wielu operacji na tym zbiorze staje się łatwiejsze.

Między innymi dotyczy to operacji:

- ▶ sprawdzenia, czy dany element, tj. element o ustalonej wartości cechy, według którego zbiór został uporządkowany, znajduje się w zbiorze;
- ▶ znalezienia elementu w zbiorze, gdy 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 folderze znajduje się plik o podanej nazwie, rozszerzeniu, czasie utworzenia lub rozmiarze, to najpierw odpowiednio porządkujemy listę plików w programie powłokowym i wtedy na ogół znajdujemy odpowiedź natychmiast. Porządkowanie jest również podstawową operacją

wykonywaną na dużych zbiorach informacji, np. w bazach danych.

Jak podaje Donald Knuth, zobacz książkę [Knuth-3], w 1973 roku producenci komputerów szacowali, że przez ponad 25% czasu pracy komputera jest wykonywane porządkowanie. Obecnie porządkowanie danych i informacji zajmuje komputerom jeszcze więcej czasu.

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 w tej książce.

Z wyjątkiem punktu 6.3, zajmujemy się w tej książce głównie porządkowaniem liczb, dlatego problem porządkowania określiliśmy na początku rozdziału 4. właśnie dla takiej sytuacji — przypomnij sobie teraz podaną tam specyfikację tego problemu oraz założenia o operacjach wykonywanych w algorytmach jego rozwiązywania.

Problem porządkowania i zbiory uporządkowane wystąpiły już we wcześniejszych fragmentach książki — w rozdziale 4. rozważaliśmy algorytmy porządkowania kilku liczb: dwóch, trzech, czterech i pięciu, wykorzystując do tego drzewa obliczeń, a w rozdziale 5. poszukiwaliśmy szczególnych elementów w zbiorach, które można by łatwo znajdować, gdyby zbiory były uporządkowane.

W tym rozdziale omawiamy pierwsze algorytmy porządkowania, które mogą być stosowane do ciągów zawierających dowolną liczbę elementów. Dwa pierwsze algorytmy są realizacjami bardzo naturalnych pomysłów, wynikających wprost z definicji problemu porządkowania. Dla algorytmu podanego w punkcie 6.2 poczyniliśmy już nawet przygotowania w poprzednim rozdziale i tutaj wykorzystamy opracowane tam algorytmy jako podprogramy. W ostatnim punkcie omawiamy metodę, która może być zastosowana do porządkowania obiektów innych niż liczby i znajduje w praktyce nie mniejsze zastosowanie niż porządkowanie liczb, zwłaszcza w odniesieniu do elementów, które składają się z kilku części, jak na przykład słowa (składają się z pojedynczych liter), daty czy rekordy (jako typ danych, np. w takim sensie, w jakim występują w języku Pascal).

Problem porządkowania omawiamy ponownie w rozdziale 10., gdzie przedstawiamy najefektywniejsze w tej książce algorytmy dla tego problemu, wykorzystujące strategię dziel i zwyciężaj. I wreszcie w punkcie 10.4 podsumowujemy to, czego mogliśmy się dowiedzieć o porządkowaniu podczas lektury tej książki.

Polecamy prostą aplikację sieciową *Sortowanie* [Oprog], umożliwiającą demonstrację w działaniu najpopularniejszych algorytmów porządkowania, w tym większości algorytmów omówionych w tej książce.

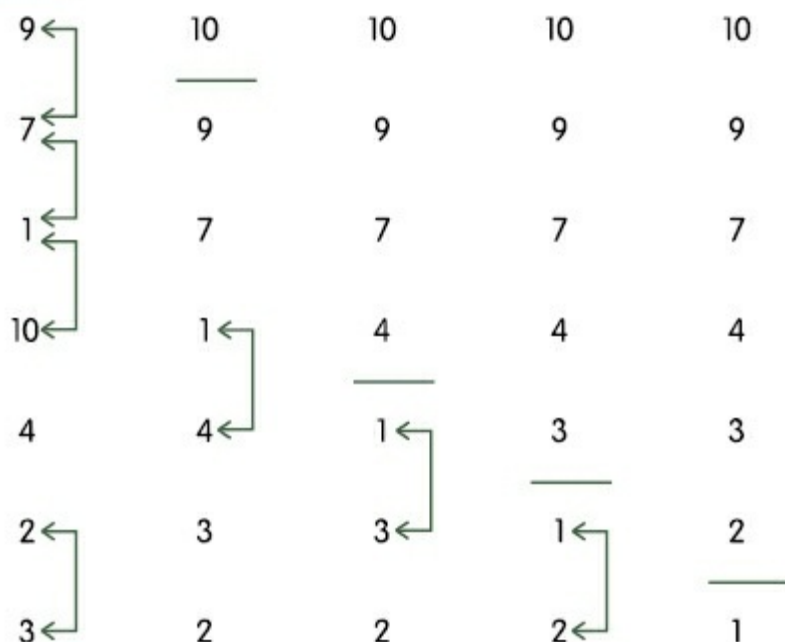
6.1. Algorytm bąbelkowy

Jeden z najprostszych koncepcyjnie algorytmów porządkowania ciągu opiera się na prostej idei: jeśli ciąg nie jest uporządkowany, to znajdują się w nim przynajmniej dwa elementy, które nie są na właściwych miejscach. Przystawmy więc te elementy i postępujemy tak dalej, aż wszystkie elementy będą na właściwych miejscach.

Można nieco uprościć to postępowanie. Zauważmy, że: jeśli ciąg zawiera dwa elementy, które nie są jeszcze ustawione we właściwej kolejności, to istnieją w nim dwa elementy, które stoją obok siebie i nie są we właściwej kolejności. Na przykład w ciągu (2, 4, 6, 1, 3, 5) elementy 4 i 3 nie są na właściwym miejscu, ale istnieją dwa elementy 6 i 1 stojące obok siebie, które również nie są odpowiednio uporządkowane. Z kolei po ich przestawieniu elementy sąsiednie 4 i 1 oraz 6 i 3 znajdą się na niewłaściwych pozycjach. Zatem w ciągu nieuporządkowanym wystarczy przestawiać ze sobą elementy, które są ustawione w złym porządku i znajdują się na sąsiednich miejscach. Aby tę strategię porządkowania zamienić na algorytm, musimy określić dodatkowo, w jakim porządku będziemy szukać takich par do przestawienia. Z jednej strony bowiem nie możemy pominąć żadnej z nich, a z drugiej — chcemy to robić w możliwie najbardziej efektywny sposób.

Algorytm bąbelkowy jest metodą porządkowania ciągów, która polega na przestawianiu sąsiednich par elementów stojących w nieodpowiedniej kolejności, przy czym ciąg jest przeglądany w tym samym kierunku tak długo, jak długo może w nim wystąpić jeszcze para elementów w niewłaściwym porządku. Oznacza to, że po każdym przestawieniu dwóch elementów należy jeszcze sprawdzić, czy ta zamiana nie zepsuła właściwego uporządkowania między innymi sąsiednimi elementami.

Działanie algorytmu bąbelkowego ilustrujemy na rysunku 6.1 w taki sposób, w jaki zwykle jest on przedstawiany — elementy o większych wartościach są wynoszone wyżej, jakby zostały do nich przywiązane większe bąbelki powietrza — stąd nazwa algorytmu. Pokazany na tym rysunku ciąg jest porządkowany w czterech etapach-przebiegach ciągu w poszukiwaniu elementów do przestawienia. Zielonymi klamrami oznaczyliśmy zamiany elementów (znaczenie zielonych linii wyjaśnimy za chwilę). Ten przykład ilustruje, że wraz z liczbą kroków zwiększa się liczba elementów, które stoją na właściwym miejscu.



Rysunek 6.1. Przykład działania algorytmu bąbelkowego

Ćwiczenie 6.1. Wykaż, że w pierwszym przebiegu algorytmu bąbelkowego przez porządkowany ciąg największy element tego ciągu — bez względu na to, gdzie się znajduje — jest umieszczany na końcu (u góry) ciągu. ■

Z treści tego ćwiczenia wynika, że w kolejnym etapie tego algorytmu mamy przynajmniej o jeden element mniej do przestawiania. Ale czy tylko jeden? Ponownie spójrzmy na rysunku 6.1. Podany przykład ilustruje, że może ich ubyc znacznie więcej. Postarajmy się to wykorzystać. Zauważmy, że jeśli (x, y) jest ostatnią parą sąsiednich liczb przestawianych w danym przebiegu algorytmu, to w następnym przebiegu nie musimy już sprawdzać par stojących powyżej elementu y — jedynie ten element może być jeszcze przestawiony niżej. I właśnie zieloną linią jest oznaczone na rysunku 6.1 to miejsce w porządkowanym ciągu, powyżej którego nie musimy już porównywać par elementów (w opisie tego algorytmu, podanym niżej, miejsce tej linii określa wartość zmiennej $Kres$).

W każdym kroku algorytmu zielona linia przesuwana się przynajmniej o jedną pozycję niżej, a więc ciąg o n elementach algorytm przebiega co najwyżej n razy, zanim ta linia nie znajdzie się na samym dole. W przykładzie pokazanym na rysunku 6.1 po drugim przebiegu algorytmu zyskujemy dwie dodatkowe pozycje, a więc zamiast 7 iteracji algorytm wykonuje tylko 5.

Podaliśmy już wszystkie szczegóły potrzebne do ścisłego opisu algorytmu bąbelkowego.

Algorytm bąbelkowy porządkowania ciągu liczb

Dane: Liczba naturalna n i ciąg n liczb x_1, x_2, \dots, x_n .

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Krok 1. $Kres := n$. { $Kres$ określa miejsce w ciągu, oznaczone narysunku 6.1 zieloną linią, stanowiące granicę poszukiwania pary elementów do przestawienia.}

Krok 2. Przyjmij $i := 1$ oraz $k := 0$ { k jest wskaźnikiem przestawianej pary.}

Krok 3. Dopóki $i < Kres$, wykonuj: jeśli $x_i > x_{i+1}$, to przestaw te dwa elementy i przyjmij $k := i$; zwiększ i : $i := i + 1$.

Krok 4. Jeśli $k > 1$, to przyjmij $Kres := k$ i wróć do kroku 2., w przeciwnym razie zakończ algorytm. ■

Ćwiczenie 6.2. Zastosuj algorytm bąbelkowy do ustawienia ciągu liter KSIĄŻKAOALGORYTMACH w porządku alfabetycznym. ■

Uwaga do ćwiczenia 6.2 (i do ćwiczeń: 6.4, 10.1, 10.5, 10.6, 10.7). Te ćwiczenia należy wykonać na papierze, nie posługując się żadnym programem. W tym celu przygotuj pokratkowany papier, zawierający przynajmniej 19 kolumn (tyle, ile liter jest w tym ciągu). Wpisz ten ciąg w pierwszym wierszu, a w następnych — wpisz kolejno wykonywane operacje i postacie tego ciągu. Na każdą iterację algorytmu przeznacz dwa wiersze — w pierwszym zaznacz operacje wykonywane na bieżącym ciągu, a w następnym — postać porządkowanego ciągu po wykonaniu tych operacji. Taki raport z wykonania algorytmu może również posłużyć do określenia liczby iteracji oraz liczby operacji wykonanych w algorytmie. W przypadku algorytmu bąbelkowego wiersze potraktuj jako kolumny. ■

Poniżej przedstawiamy programy w językach Pascal i Python, realizujące algorytm bąbelkowy. Podobnie, jak w słownym opisie tego algorytmu, ważna jest redukcja liczby działań poprzez pamiętanie miejsca, powyżej którego ciąg jest już uporządkowany. Poświęcona jest temu druga część ćwiczenia 6.3.



```
procedure BubbleSort(n:integer; var x:Tablica1n);  
  var i,k,Kres,r:integer;  
begin  
  Kres:=n;  
  while Kres>1 do begin  
    k:=1;  
    for i:=1 to Kres-1 do  
      if x[i]>x[i+1] then begin  
        r:=x[i]; x[i]:=x[i+1]; x[i+1]:=r;  
        k:=i
```



```

end;
Kres:=k
end {while}
end; {BubbleSort}

```



```

def BubbleSort(lista):
    Kres=len(lista)-1
    while Kres > 0:
        k=0;
        for i in range(0,Kres,1):
            if lista[i]>lista[i+1]:
                r=lista[i]
                lista[i]=lista[i+1]
                lista[i+1]=r
                k=i
        Kres=k

```

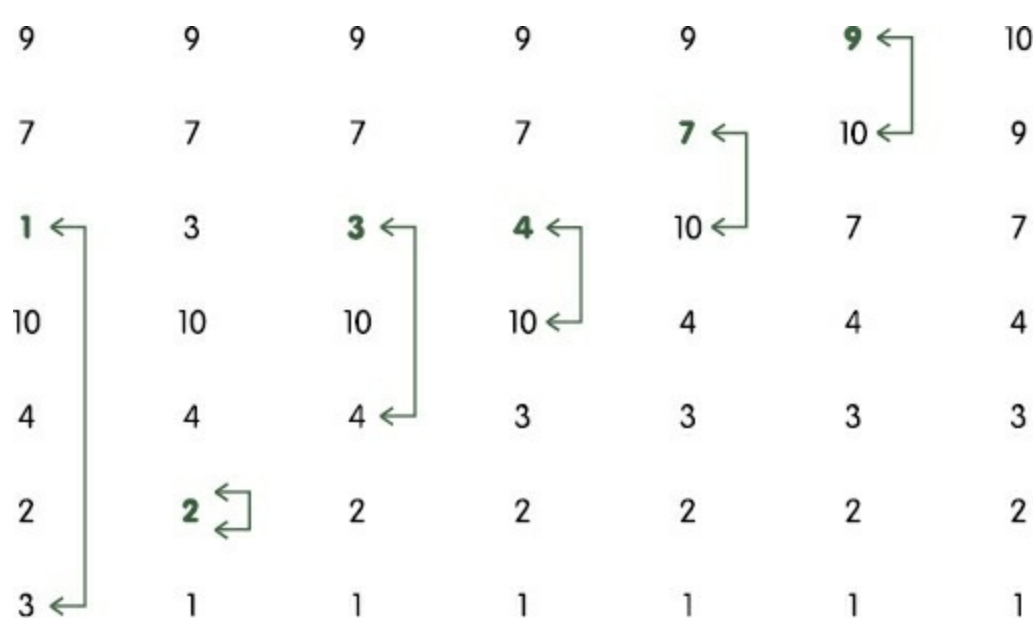
Ćwiczenie 6.3. Podaj przykład ciągu, dla którego algorytm bąbelkowy wykonuje największą możliwą liczbę porównań i przestawień elementów. Ile wynosi ta liczba — czy potrafisz ją określić w zależności od liczby elementów w ciągu? A dla jakiego ciągu liczb algorytm bąbelkowy wykonuje najmniejszą liczbę porównań i zamian elementów miejscami? Ile wynosi ta liczba w zależności od liczby elementów w posortowanym ciągu? Sprawdź swoją odpowiedź na przykładach, posługując się programami w języku Pascal lub Python, wcześniej modyfikując te programy, by zliczały liczby wykonywanych porównań i zamian.

6.2. Porządkowanie przez wybór

Algorytm opisany w tym punkcie można, podobnie jak algorytm bąbelkowy, wyprowadzić niemal z definicji problemu porządkowania. Zauważmy, że: jeśli mamy ustawić elementy ciągu w kolejności od najmniejszego do największego, to można wybrać w nim najpierw element najmniejszy i umieścić go na początku, za nim umieścić drugi najmniejszy element ciągu (czyli najmniejszy w pozostałym ciągu) itd. Ta metoda nazywa się **algorytmem porządkowania przez wybór**. Do jej dokładnego opisu jesteśmy już przygotowani, gdyż poznaliśmy szybką (wręcz optymalną) metodę znajdowania najmniejszego elementu w ciągu (zobacz punkt 5.3). Musimy jedynie podać, w jaki sposób kolejno znajdowane elementy, od najmniejszego do największego, mają być

ustawiane jeden za drugim. Najoszczędniej byłoby robić to w tym samym miejscu, w którym jest zapisany ciąg do uporządkowania. Jest to możliwe: znaleziony element zamienia się miejscami z elementem, który zajmuje jego miejsce w uporządkowanym ciągu. A więc po znalezieniu najmniejszego elementu trzeba zamienić go miejscami z pierwszym elementem ciągu, w następnym etapie — drugi najmniejszy element w ciągu zamienić miejscami z drugim elementem ciągu itd. Zauważmy, że w takiej realizacji tej metody wystarczy znajdować miejsce (a dokładniej — jego indeks) w ciągu zajmowane przez coraz większe elementy.

Sposób działania algorytmu porządkowania przez wybór ilustrujemy na rysunku 6.2 na przykładzie ciągu z rysunku 6.1. Zwróćmy uwagę na drugą iterację — bieżący najmniejszy element może się już znajdować na swoim miejscu w ciągu. W realizacji tego algorytmu nie wyróżniamy jednak specjalnie tej sytuacji — w tym przypadku taki element jest zamieniany miejscem z samym sobą.



Rysunek 6.2. Przykład działania algorytmu porządkowania przez wybór

Przedstawmy teraz ścisły opis tego algorytmu.

Algorytm porządkowania przez wybór

Dane: Liczba naturalna n i ciąg n liczb x_1, x_2, \dots, x_n .

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Krok 1. Dla $i = 1, 2, \dots, n - 1$ wykonaj kroki 2. i 3.

Krok 2. Znajdź k takie, że x_k jest najmniejszym elementem w podciągu x_i, \dots, x_n .

Krok 3. Zamień miejscami elementy x_i oraz x_k . ■

Ćwiczenie 6.4. Zastosuj algorytm porządkowania przez wybór do ustawienia ciągu liter KSIĄŻKAOALGORYTMACH w porządku alfabetycznym. (Zobacz

uwagę zapisaną po ćwiczeniu 6.2 w punkcie 6.1). ■

Poniżej przedstawiamy realizację algorytmu porządkowania przez wybór w obu językach programowania.



```
procedure SelectionSort (n:integer; var x:Tablica1n);
  var i,j,k,r:integer;
begin
  for i:=1 to n-1 do begin
    k:=i;
    for j:=i+1 to n do
      if x[j]<x[k] then k:=j
    r:=x[i]; x[i]:=x[k]; x[k]:=r;
  end {for}
end; {SelectionSort}
```



```
def SelectionSort(lista):
  for i in range(0,len(lista)-1,1):
    k=i;
    for j in range(i+1,len(lista),1):
      if lista[k]>lista[j]:
        k=j
    r=lista[i]
    lista[i]=lista[k]
    lista[k]=r
```

Ćwiczenie 6.5. Prześledź działanie zamieszczonych wyżej programów na ciągach o różnej strukturze elementów, w tym na ciągach uporządkowanych i odwrotnie uporządkowanych. Przekonaj się, że przy ustalonej liczbie elementów w ciągu liczby porównań i zamian elementów operacji nie zależą od wartości porządkowanych elementów. Uzasadnij to stwierdzenie. ■

Polecamy prosty program edukacyjny *Maszyna sortująca* ze strony [Oprog], w którym można bardzo szczegółowo prześledzić działanie algorytmu porządkowania przez wybór. Ten program może być pomocny przy udzielaniu odpowiedzi w ćwiczenia 6.5. Możesz skorzystać także z aplikacji sieciowej *Sortowanie* [Oprog].

Złożoność

Aby znaleźć postać wyrażenia określającego liczbę działań (porównań i przestawień elementów) wykonywanych w algorytmie porządkowania przez wybór, wystarczy zauważyć, że jest on 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, a więc wynosi $n - 1$. Jeśli zaś chodzi o liczbę porównań, to w punkcie 5.3 podaliśmy, że w algorytmie znajdowania minimum w ciągu wykonujemy o jedno porównanie mniej niż jest elementów w ciągu. Ponieważ w każdym kroku liczba elementów w przeszukiwanym podciągu jest o jeden mniejsza, zatem w algorytmie porządkowania przez wybór, dla ciągu danych złożonego z n elementów, liczba porównań wynosi:

$$(n - 1) + (n - 2) + \dots + 2 + 1 \quad (6.1)$$

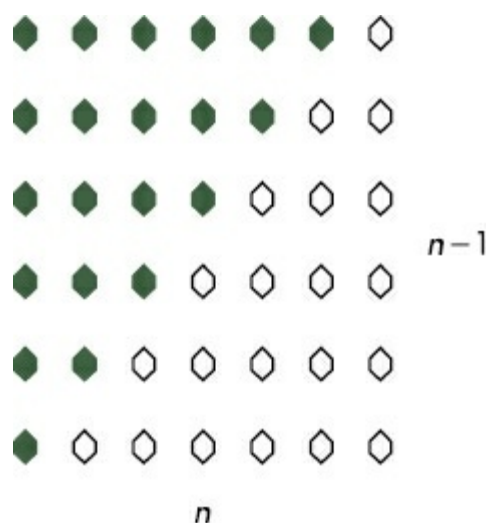
Wartość sumy (6.1) można obliczyć wieloma sposobami. Przedstawiamy dwa z nich — są one ciekawe przez swoją prostotę. Pomijamy tutaj inny sposób, w którym korzysta się ze wzoru na sumę ciągu arytmetycznego, jako oczywisty dla tych, którzy wiedzą, co to jest ciąg arytmetyczny.

Ten geometryczny dowód znali już starożytni Grecy. Liczby mające postać (6.2) nazywali **liczbami trójkątnymi**. Zobacz problem 6.1 o liczbach kwadratowych.

Dowód geometryczny

Kolejne liczby naturalne od 1 do $n - 1$ można przedstawić w postaci trójkąta, którego wiersz i zawiera i diamentów (na rysunku 6.3 są to wypełnione diamenty). Dwa takie same trójkąty pasują do siebie i tworzą prostokąt zawierający $(n - 1)n$ diamentów, zatem wartość sumy (6.1) jest połową liczby wszystkich diamentów w całym prostokącie, czyli jest równa:

$$\frac{(n-1)n}{2} \quad (6.2)$$



Rysunek 6.3. Ilustracja geometrycznego wyznaczania wartości sumy (6.1)

Geniusz — gen i już.

Hugo Steinhaus

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 $1 + 2 + 3 + \dots + 98 + 99 + 100$. Nie cieszył się jednak zbyt długo spokojem, po chwili bowiem otrzymał 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 otrzymujemy wzór (6.2) dla $n - 1$ kolejnych liczb w ciągu.

Ćwiczenie 6.6. Przekonaj się, że w rozumowaniu C. F. Gaussa nie ma błędu — jest ono poprawne bez względu na to, czy ilość sumowanych liczb we wzorze (6.1) jest parzysta czy nieparzysta. ■

6.3. Porządkowanie kubelkowe i pozycyjne

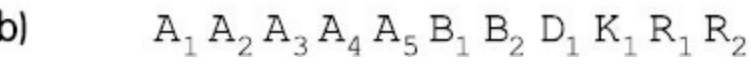
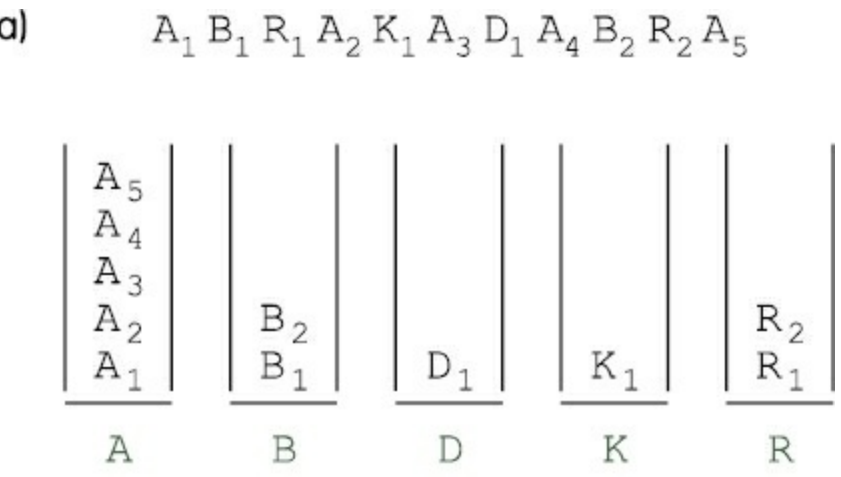
6.3.1. Porządkowanie kubelkowe

Jednym z najprostszych sposobów porządkowania jest technika stosowana przy sortowaniu listów. Jeśli miałeś kiedyś okazję widzieć sortującego listy urzędnika poczty, to zauważyłeś, że listy, które trzyma w ręce, rozkłada on do różnych przegródek. O tym, do której z nich trafi koperta, decydują różne fragmenty adresu. Dla listów zagranicznych są przeznaczone przegródki oznaczone nazwami państw; dla listów krajowych zamiejscowych — przegródki z nazwami województw, a dla listów miejscowych — z nazwami dzielnic. Na ogół liczba przegródek jest niewielka, od 10 do 50. Niestety, przy obecnie obowiązującym systemie oznaczania adresów kodami ten sposób sortowania przesyłek krajowych jest bardzo uciążliwy. Z jednej strony bowiem zniknęły z adresów nazwy województw, a z drugiej — liczba różnych kodów jest tak duża, że nie można pozwolić sobie na sortowanie tą metodą, przyjmując za „nazwę przegródki” numer kodowy odbiorcy.

Naszkicowana metoda nazywa się **porządkowaniem kubelkowym** lub **koszykowym**, w zależności od tego, jak są nazwane pojemniki, do których wrzuca się porządkowane elementy (obiekty). By zbiór został całkowicie uporządkowany, kubelki wypełnione w pierwszym etapie muszą być opróżniane w następnym według kolejności przypisanych im nazw, dając w efekcie kolejność

elementów rozważanego zbioru.

Zastosujmy teraz algorytm kubełkowy do uporządkowania liter tworzących słowo ABRAKADABRA. To słowo ma pięć różnych liter: A, B, D, K i R. Zakładamy więc pięć kubełków, oznaczamy je tymi literami i ustawiamy w takiej kolejności, w jakiej te litery występują w alfabecie — zobacz rysunek 6.4a. W pierwszym kroku algorytmu przeglądamy to słowo i rozdzielamy jego litery do kubełków, a w drugim kroku tworzymy już uporządkowany ciąg, opróżniając kubełki w kolejności ich ustawienia. Na rysunku 6.4a kolejne wystąpienia tej samej litery są oznaczone kolejnymi indeksami, by można było łatwo odnaleźć poszczególne litery w kubełkach, a później w uporządkowanym ciągu. Litery są umieszczane w kubełkach w kolejności występowania w słowie. Zauważ, że kubełki nie są domknięte od dołu — zaznaczyliśmy w ten sposób, że są one opróżniane od dołu. Dzięki temu litery opuszczają kubełki w takiej samej kolejności, w jakiej do nich trafiają. Co Ci to przypomina? Oczywiście — znaną z życia kolejkę, w której nie obowiązują żadne przywileje. **Kolejka** jest jedną z podstawowych struktur danych występujących w algorytmach. Ten sposób opróżniania kubełków gwarantuje, że poszczególne wystąpienia danej litery są w takiej samej kolejności w słowie ABRAKADABRA, jak i w uporządkowanym ciągu liter — por. rysunek 6.4a i 6.4b (w punkcie 10.4 dokładniej opisujemy tę własność algorytmów porządkowania, zwaną **stabilnością**).



Rysunek 6.4. Przykład działania algorytmu kubełkowego, porządkującego ciąg pojedynczych liter: a) pierwszy etap — umieszczanie liter w kubełkach; b) drugi etap — ciąg liter po opróżnieniu kubełków od dołu

Ćwiczenie 6.7. W ćwiczeniach 6.2 i 6.4 należało uporządkować litery w ciągu KSIĄŻKAOALGORYTMACH, korzystając z dwóch algorytmów, których głównym przeznaczeniem jest porządkowanie liczb. Zastosuj teraz algorytm kubełkowy do uporządkowania tego ciągu. Porównaj liczbę operacji wykonanych przez te trzy algorytmy. ■

6.3.2. Porządkowanie pozycyjne

W jaki sposób można rozszerzyć kubełkowe porządkowanie liter na porządkowanie całych słów? Aby porządkować słowa, należy najpierw określić, które z dwóch danych słów powinno wystąpić wcześniej. Najprościej jest przyjąć naturalny porządek między słowami, nazywany **porządkiem leksykograficznym** lub **słownikowym**, czyli taki, jaki jest w słownikach i w encyklopediach. Można go zdefiniować następująco:

► jeśli dwa słowa mają jednakową długość, to szukamy w nich pierwszej pozycji, na której się różnią, a wówczas wcześniejsze w kolejności jest to słowo, które ma na tej pozycji wcześniejszą w alfabecie literę — np. ARAB jest wcześniejsze od ARAK, gdyż B występuje w alfabecie przed K, a DRAB jest wcześniejsze niż KRAB, gdyż D poprzedza K w alfabecie;

► jeśli słowa mają nierówną długość, to albo szukamy pierwszej pozycji, na której się różnią i litera na tej pozycji decyduje o miejscu ustawienia słowa (podobnie jak w przypadku słów o równej długości) — np. słowo BAR jest wcześniejsze od BRDA, gdyż A poprzedza R, albo jedno słowo jest częścią drugiego, wtedy występuje w słowniku przed każdym słowem, w którym jest zawarte na początku — np. BAR poprzedza BARD i BARK.

W następnych podpunktach omawiamy zastosowania metody kubełkowej do słownikowego porządkowania: słów, jednakowej i różnej długości, liczb oraz dat.

Porządkowanie słów jednakowej długości

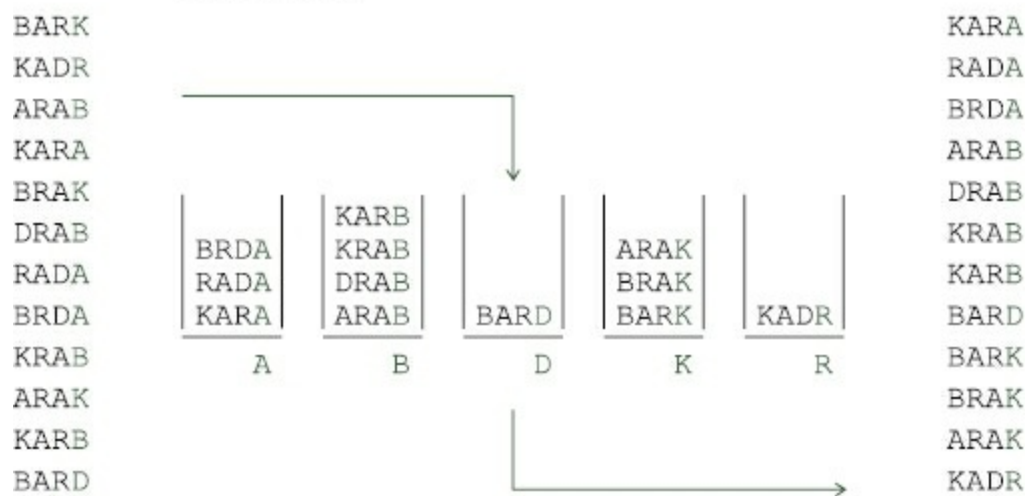
Z określenia porządku leksykograficznego wynika, że: jeśli słowa są jednakowej długości i różnią się tylko na ostatniej pozycji, to ich kolejność w słowniku jest wyznaczana przez kolejność ostatnich liter, np. słowo ARAB poprzedza ARAK, a BARD poprzedza BARK. Podobnie kolejność słów różniących się na dwóch ostatnich pozycjach jest wyznaczana przez kolejność par liter na tych pozycjach.

Ideę porządkowania pozycyjnego można odnaleźć w pomysłach Hermana Holleritha, który pod koniec XIX wieku budował maszyny do automatyzacji przetwarzania wyników spisu ludności w USA. Czytnik i sorter kart, na których były zapisane wyniki spisu, działały zgodnie z algorytmem kubełkowym, stosowanym osobno do pozycji jednostek i dziesiątek, w opisach patentów nie można jednak znaleźć uzasadnienia, w jakiej kolejności należy to robić. H. Hollerith założył firmę, która później przekształciła się w wielki koncern IBM (ang. *International Business Machines*).

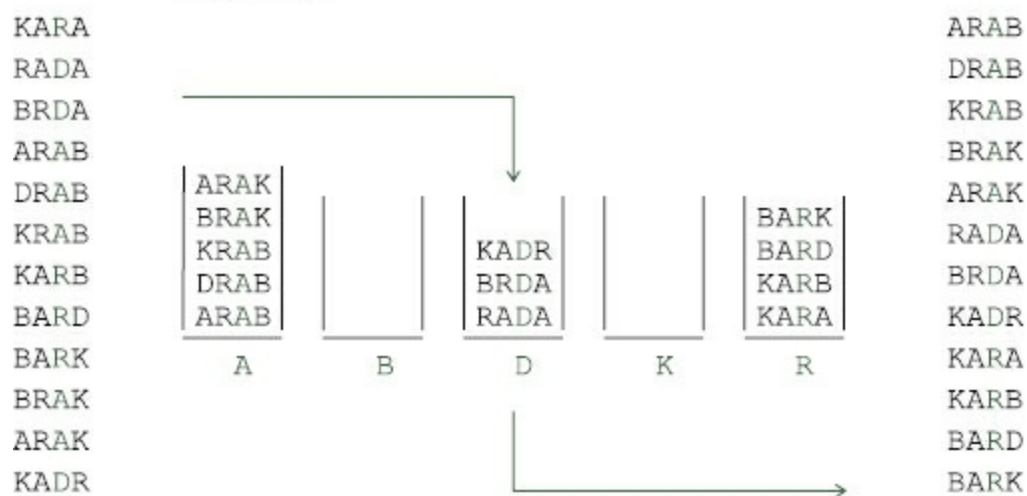
Zatem porządkowanie słów o jednakowej długości polega na porządkowaniu ich od końca, pozycja po pozycji, z zastosowaniem do każdej pozycji algorytmu kubełkowego. Taka metoda nazywa się algorytmem **porządkowania pozycyjnego**, dla podkreślenia, że porządkowane obiekty: liczby, słowa,

rekordy występują w postaci pozycyjnej i porządkowanie odbywa się względem każdej pozycji. Na rysunku 6.5 ilustrujemy sposób działania algorytmu pozycyjnego na zbiorze słów utworzonych z czterech liter, które zostały wybrane ze słowa ABRAKADABRA. Iteracje odpowiadają pozycjom, względem których odbywa się porządkowanie. Każda iteracja jest jednym przebiegiem algorytmu kubełkowego dla liter stojących we wszystkich słowach na tej samej pozycji, względem której odbywa się porządkowanie. Zielonym kolorem jest oznaczona pozycja i litery, które są uwzględniane w danej iteracji. W każdej iteracji, po lewej stronie kubełków jest pokazany ciąg słów przed wykonaniem algorytmu kubełkowego dla zaznaczonej pozycji, a po prawej — po opróżnieniu kubełków. Zauważmy, że w niektórych iteracjach niektóre kubełki są puste.

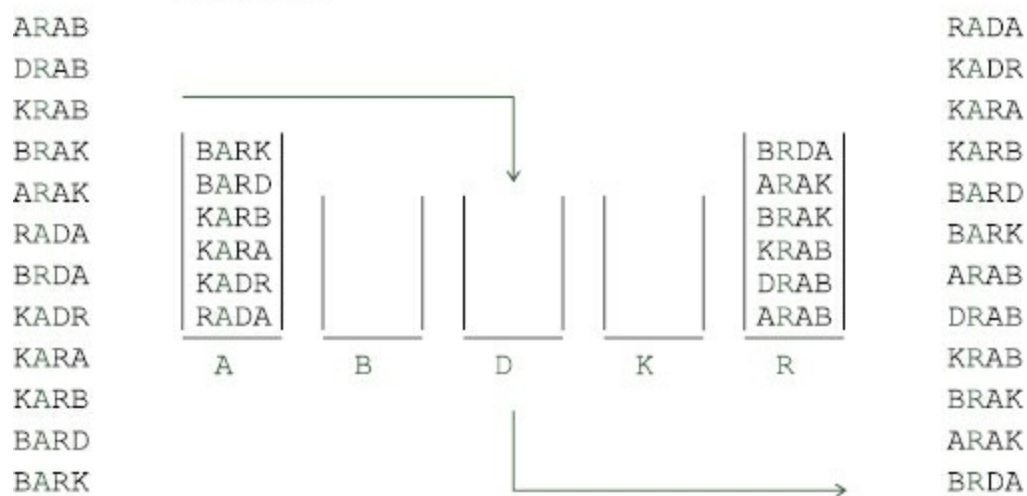
Pierwsza iteracja:



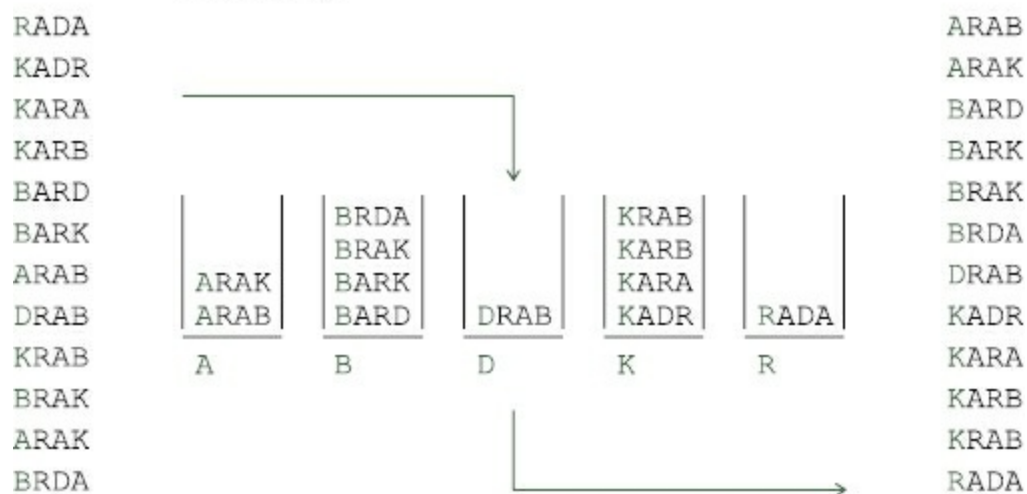
Druga iteracja:



Trzecia iteracja:



Czwarta iteracja:



Rysunek 6.5. Przykład działania algorytmu porządkowania pozycyjnego na zbiorze słów czteroliterowych. Zielonym kolorem jest oznaczona pozycja w słowach, względem której odbywa się w danej iteracji porządkowanie metodą kulek.

Porządkowanie słów nierównej długości

W jaki sposób możemy teraz rozbudować algorytm porządkowania pozycyjnego, aby były porządkowane również słowa o różnych długościach? Z definicji porządku słownikowego wynika, że również w tym przypadku słowa należy przeglądać od ostatnich liter. Pozostaje więc jedynie rozstrzygnąć, gdzie względem dłuższych słów należy umieścić słowa krótsze. Z drugiej części definicji porządku wynika, że o kolejności słów różnej długości decydujemy, porównując je pozycja po pozycji, przy czym zaczynamy od pierwszej z lewej. Zatem w algorytmie pozycyjnym najpierw powinny być zrównane pierwsze litery wszystkich słów, później — drugie itd. Można więc powiedzieć, że wszystkie słowa należy dosunąć do lewej strony — zobacz pierwszą kolumnę słów na rysunku 6.6. A co należy zrobić z krótszym słowem, gdy porządkujemy słowa względem pozycji, która wykracza poza jego ostatnią literę z prawej strony? W tym przypadku po prostu dopisujemy słowo do tworzonego w danej iteracji ciągu

wszystkich słów, do którego po zakończeniu wypełniania kubelków będziemy dopisywać słowa z opróżnianych kubelków. Dzięki temu krótsze słowa będą poprzedzały w uporządkowaniu dłuższe słowa je zawierające, tak jak to jest w słownikach.

DRAKA	DRAKA	DRAKA	RA	RA	RA	RA	ARA
ARABKA	ARABKA	RA	ARA	ARA	ARA	RADAR	ARAB
RA	RA	ARA	KRA	KRA	KRA	BAR	ARABKA
ARA	ARA	KRA	ARAB	BAR	ARAB	BARBARA	BAR
BARBARA	KRA	ARAB	BARD	RADAR	ARABKA	BARD	BARBARA
KRA	ARAB	BARD	BAR	ARAB	DRAKA	ARA	BARD
ARAB	BARD	BAR	DRAKA	BARBARA	RADAR	KRA	DRAKA
BARD	BAR	RADAR	BARBARA	ARABKA	BAR	ARAB	KRA
BAR	RADAR	ARABKA	ARABKA	BARD	BARBARA	ARABKA	RA
RADAR	BARBARA	BARBARA	RADAR	DRAKA	BARD	DRAKA	RADAR

Czy potrafisz odgadnąć, jaki jest porządek wyrazów w pierwszej kolumnie na rysunku 6.6, oraz w jaki sposób można znaleźć takie uporządkowanie? Nie jest to uporządkowanie przypadkowe.

Rysunek 6.6. Przykład działania algorytmu porządkowania pozycyjnego na zbiorze słów o różnej długości. Pokazane są postacie ciągu słów po każdej iteracji algorytmu kubelkowego, zastosowanego do kolejnych pozycji w słowach. Jasnozielonym kolorem wyróżniono litery, według których będzie przebiegać następna iteracja, a ciemnozielonym — te same litery w następnej kolumnie już po uporządkowaniu słów względem nich

Na rysunku 6.6 ilustrujemy działanie algorytmu porządkowania pozycyjnego na przykładzie ciągu wybranych słów różnej długości, utworzonych z liter występujących w słowie ABRAKADABRA. Każda kolejna kolumna, z wyjątkiem pierwszej, zawiera słowa uporządkowane algorytmem kubelkowym, zastosowanym do pozycji oznaczonej jasnozielonym kolorem w poprzedniej kolumnie. Ciemnozielonym kolorem są oznaczone litery po uporządkowaniu. Zgodnie z opisem tego algorytmu, krótsze słowa, czyli takie, które nie zawierają litery na pozycji rozważanej w danej iteracji, są cyklicznie przepisywane na koniec ciągu słów podczas jego przeglądania. Natomiast słowa zawierające jasnozielone litery są umieszczane w odpowiednich kubelkach. Po przejrzaniu całego ciągu kubelki są opróżniane (od dołu), a ich zawartość jest dopisywana do końca ciągu — litery w tych słowach, według których trafiały one do odpowiednich kubelków, wyróżniono ciemnozielonym kolorem.

Pozycyjne porządkowanie liczb

Zacznij od uporania się z ćwiczeniem, w którym masz uporządkować wielocyfrowe liczby. Porównywanie wartości tak dużych liczb jest dość uciążliwe — jak sobie z tym poradzisz?

Ćwiczenie 6.8. Zapisz na oddzielnych kartkach 20 liczb, składających się z

sześciu, siedmiu lub ośmiu cyfr. Jak szybko uporządkujesz te kartki zgodnie z wartościami tych liczb? Potasuj je i daj do uporządkowania swojemu koledze. Ile czasu jemu to zabrało? Daj to zadanie kilku innym osobom. Opisz metodę, która okazała się najszybsza. ■

Nawet wtedy, gdy wykonując to ćwiczenie, porównywałeś wartości liczb, to jednak ze względu na dużą długość liczb i różną w nich liczbę cyfr, musiałeś wiele razy porównywać pojedyncze cyfry. A gdyby tak zastosować algorytm porządkowania pozycyjnego?

24	105
135	135
5	24
279	279
37	37
105	5

Zapewne otrzymałeś ciąg tych liczb w takim porządku, w jakim występują po prawej stronie tekstu ćwiczenia. Nie są one jednak ustawione według wartości — a jaka jest ich kolejność? Zauważ, że taka sama, jak słów w słowniku, przy założeniu, że kolejność „liter alfabetu”, którymi w tym przypadku są cyfry, jest zgodna z ich wartościami, czyli 0, 1, 2 itd. Czy w takim razie porządkowania pozycyjnego nie można stosować do porządkowania liczb? Można, ale z jedną modyfikacją. Spójrz najpierw na otrzymane rozwiązanie i zastanów się, dlaczego liczby 135, 24 i 5 zostały ustawione w tej kolejności. Powód jest widoczny — zdecydowały o tym cyfry stojące na pierwszej pozycji, które jednak dla każdej z tych liczb mają inne znaczenie: w pierwszej jest to liczba setek, w drugiej — liczba dziesiątek, a w trzeciej — liczba jedności. Czyli w tym przypadku algorytm porządkowania pozycyjnego w jednej iteracji badał cyfry w liczbach znajdujące się na pozycjach, które nie powinny być porównywane, gdy szukamy kolejności liczb względem ich wartości. Można to łatwo „naprawić”, ustawiając liczby w taki sposób, aby w tej samej kolumnie znalazły się cyfry jedności, w następnej — cyfry dziesiątek, w kolejnej — cyfry setek itd. W tym celu wystarczy dosunąć wszystkie liczby do prawej strony.

24	5
135	24
5	37
279	105
37	135

Potrafiemy więc już porządkować liczby algorytmem pozycyjnym. Metoda ta nie jest jednak wygodna do porządkowania liczb, gdy są one pamiętane w komputerze w postaci binarnej, ponieważ wtedy nie ma łatwego dostępu do ich cyfr w rozwinięciu dziesiętnym. Na przykład liczba 5 ma rozwinięcie binarne 101 i wszystkie te cyfry binarne muszą być uwzględnione, by stwierdzić, że jest to rzeczywiście liczba 5. Algorytm porządkowania pozycyjnego jest natomiast bardzo dogodną metodą, gdy kolejne cyfry porządkowanych liczb są pamiętane w oddzielnych komórkach (np. w bajtach). W taki sposób można na przykład reprezentować pocztowe numery kodowe.

Porządkowanie dat

Algorytm porządkowania pozycyjnego ma wiele zastosowań w praktyce. Był znany i stosowany na długo przed skonstruowaniem pierwszych komputerów. Przede wszystkim jest odpowiedni do porządkowania elementów (obiektów, wielkości), w których można wyróżnić pozycje, a porządek między elementami opisać, posługując się porządkiem wielkości zajmujących te same pozycje. Innymi obiektami, które można porządkować algorytmem pozycyjnym, są daty. Wykonaj najpierw następujące ćwiczenie, które powinno naprowadzić Cię na trop właściwego algorytmu.

Ćwiczenie 6.11. Ćwiczenie wspólne dla całej klasy. Przygotuj kartki jednakowej wielkości i rozdaj wszystkim uczniom. Niech każdy napisze na kartce swoją datę urodzenia w postaci dd.mm.yyyy. Następnie zbierz kartki w dowolnej kolejności. W jaki sposób je uporządkujesz od najwcześniejszej do najpóźniejszej daty? ■

Najważniejszy wniosek nasuwający się po wykonaniu tego ćwiczenia powinien brzmieć: daty, chociaż są złożone z cyfr, nie mogą być porządkowane jako pojedyncze liczby, a ich kolejność zależy od wartości odpowiadających dniom, miesiącom i latom. Zatem najodpowiedniejsza będzie metoda porządkowania pozycyjnego. Pozostaje jedynie ustalić, w jakiej kolejności należy rozpatrywać poszczególne pozycje dat. To jednak nie powinno nastręczyć większych kłopotów. Bez względu na dzień i miesiąc, z dwóch dat jedna jest wcześniejsza od drugiej, gdy ma wcześniejszy rok. Jeśli w datach lata są takie same, to o ich kolejności decydują miesiące. Najmniejsze znaczenie mają dni. Zgodnie z zasadą leżącą u podstaw poprawności tej metody, głoszącą, że porządkowanie jest wykonywane w kierunku od najmniej do najbardziej znaczącej pozycji, gdy porządkowane są daty — kolejne iteracje dotyczą: dni, miesięcy i lat. To ustalenie ma istotne znaczenie, gdyż nie istnieje jedna powszechnie przyjęta kolejność ustawiania w datach dnia, miesiąca i roku. Na rysunku 6.7 przedstawiono przykład zastosowania algorytmu pozycyjnego do uporządkowania kilku nieprzypadkowych dat.

31.01.1997	03.11.1995	31.01.1997	24.09.1994
13.09.1996	10.04.1995	10.04.1995	10.04.1995
03.11.1995	13.09.1996	13.09.1996	03.11.1995
17.09.1997	15.12.1995	17.09.1997	15.12.1995
10.04.1995	17.09.1997	24.09.1994	13.09.1996
30.10.1996	24.09.1994	30.10.1996	30.10.1996
15.12.1995	30.10.1996	03.11.1995	31.01.1997
24.09.1994	31.01.1997	15.12.1995	17.09.1997

Rysunek 6.7. Przykład porządkowania kilku dat. Znaczenie kolorów jest podobne jak na rysunku 6.6: jasnozielonym kolorem wyróżniono pozycję, według której przebiega następna iteracja, a ciemnozielonym — tę samą pozycję w następnej kolumnie już po uporządkowaniu dat względem niej

6.3.3. Realizacja porządkowania pozycyjnego

Podamy teraz ścisły opis algorytmu porządkowania pozycyjnego, który ze względu na różnorodność zastosowań — trzy z nich: porządkowanie słów, liczb i dat opisaliśmy powyżej — jest na tyle ogólny, że można go dostosować do każdego z tych i innych przypadków.

Algorytm porządkowania pozycyjnego

Dane: Ciąg elementów, z których każdy jest układem pozycji wypełnionych znakami, dozwolonymi dla poszczególnych pozycji.

Wynik: Uporządkowanie danego ciągu zgodnie z leksykograficzną relacją, zdefiniowaną odpowiednio do kolejności pozycji i porządku znaków, które mogą wypełniać poszczególne pozycje.

Krok 1. Dla kolejnych pozycji w porządkowanych elementach, w kolejności od najmniej znaczącej do najbardziej znaczącej, wykonaj kroki 2., 3. i 4.

Krok 2. Załóż kubeczki dla każdego możliwego znaku, jaki może wystąpić na bieżącej pozycji w porządkowanych elementach. Ustaw te kubeczki zgodnie z porządkiem tych znaków.

Krok 3. Dla wszystkich elementów z porządkowanego ciągu wykonaj: jeśli rozważana pozycja w elemencie jest pusta, to dopisz ten element do końca porządkowanego ciągu, a w przeciwnym razie — włóż ten element do kubeczka oznaczonego znakiem stojącym na rozważanej pozycji.

Krok 4. Dla kolejnych kubeczków: przenieś elementy z kubeczka na koniec porządkowanego ciągu w takiej samej kolejności, w jakiej były w nim umieszczane.

Odpowiednią wersję opisu tego algorytmu do porządkowania słów otrzymujemy po zastąpieniu wyrazu „element” wyrazem „słowo”, natomiast po zastąpieniu

znaków na poszczególnych pozycjach — literami. Podobnie w przypadku liczb. Z kolei dla dat „znakami” na poszczególnych pozycjach są dni (od 1 do 31), miesiące (od 1 do 12) i lata z okresu, do którego należą porządkowane daty.

Ćwiczenie 6.12. Uzupełnij szczegóły w ogólnym opisie algorytmu porządkowania pozycyjnego dla wybranego przez siebie rodzaju elementów: słów, liczb lub dat. ■

Nie omawiamy tutaj szczegółowo żadnej realizacji algorytmu porządkowania pozycyjnego w postaci programu w języku Python lub Pascal. Ze względu na złożoność operacji i różnorodność elementów te realizacje są bardziej złożone. Dodatkowo efektywne realizacje tego algorytmu wymagają użycia dynamicznych struktur danych, z których nie korzystamy w tej książce.

W zasobach internetowych do tej książki zamieszczamy jednak programy korzystające z bardziej zaawansowanych struktur danych. Jednym z nich jest procedura `KubelkiLiczb` w języku Pascal porządkująca liczby. Liczby w tej procedurze są reprezentowane przez wektory (czyli jednowymiarowe tablice) swoich cyfr, a ich ciąg dany do uporządkowania jest listą połączoną, zrealizowaną za pomocą typu wskaźnikowego. Kubełki są kolejkami, czyli listami utworzonymi również za pomocą typu wskaźnikowego. W takiej realizacji operacje przemieszczania elementów i opróżniania kubełków są wykonywane za pomocą zmiany wartości wskaźników i ich efektywność nie zależy od liczby elementów w kubełkach (w listach).

6.3.4. Złożoność porządkowania pozycyjnego

Zarówno w algorytmach porządkowania kilku liczb (rozdział 4.), jak i w algorytmach porządkowania bąbelkowego oraz przez wybór (punkt 6.1, punkt 6.2), podstawową operacją wykonywaną na elementach jest ich porównywanie parami. Dlatego ocenę efektywności tych algorytmów pod względem liczby wykonywanych operacji podajemy w zależności od liczby porównań.

W algorytmach porządkowania kubełkowego i pozycyjnego nie są wykonywane żadne operacje porównywania między porządkowanymi elementami. Podstawową operacją jest natomiast przemieszczanie elementu z porządkowanego ciągu do kubełka (w kroku 3.) lub na koniec ciągu (w krokach 3. i 4.). Analizując przykład realizacji tego algorytmu w języku Pascal (w procedurze `KubelkiLiczb`, zobacz również problem 6.4), można się przekonać, że w rzeczywistości nie trzeba przenosić elementów na koniec porządkowanego ciągu — wystarczy je pozostawić tam, gdzie są, i nie trzeba pojedynczo przenosić elementów z kubełków do ciągu — wystarczy „podczepiać” całe zawartości kubełków do tworzonego ciągu.

Oznaczmy przez l maksymalną długość porządkowanych elementów, przez mi — liczbę różnych znaków, jakie mogą wystąpić na pozycji i w porządkowanych

elementach ($i = 1, 2, \dots, l$), przez n — liczbę porządkowanych elementów, a przez k_j — liczbę znaków w j -tym elemencie ($j = 1, 2, \dots, n$). Z dyskusji powyżej wynika, że w algorytmie porządkowania pozycyjnego wykonujemy:

► $K = k_1 + k_2 + \dots + k_n$ umieszczeń porządkowanych elementów w kubekach, gdyż każdy element jest lokowany tyle razy, ile zawiera znaków;

► $M = 2(m_1 + m_2 + \dots + m_l)$ operacji na kubekach: otwierania kubków w kroku 2. i dołączania kubków do ciągu porządkowanych elementów w kroku 4.

Zauważ, że algorytm pozycyjnego porządkowania jest optymalny w tym sensie, że każda pozycja w każdym elemencie jest sprawdzana tylko raz.

Założmy, że porządkujemy słowa języka polskiego. Wtedy złożoność algorytmu porządkowania pozycyjnego wynosi $K + 2 \cdot 32l$ wszystkich działań, gdzie K jest sumaryczną długością słów, a l maksymalną długością słowa. Dla uproszczenia przyjęliśmy tutaj, że na każdej pozycji w słowie może wystąpić każda z 32 liter alfabetu polskiego. Dla liczb w systemie dziesiętnym, w wyrażeniu na złożoność, 32 zastępujemy przez 10, ponieważ tyle jest możliwych cyfr dla każdej pozycji. Z kolei dla n dat to wyrażenie ma postać $3n + 2$ ($31 + 12 + p$), gdzie p jest długością czasu (czyli liczbą możliwych lat w datach), z którego pochodzą daty.

Algorytmy porządkowania pozycyjnego są najbardziej praktycznymi i najszybszymi metodami porządkowania elementów, które mają postać rekordów złożonych z pewnej liczby pozycji (zwanych polami), w których występuje niewielka liczba możliwych znaków (a ogólniej — wartości). Taką metodą porządkuje się na przykład listy plików względem różnych pozycji: nazwy, rozszerzenia, daty utworzenia, wielkości.

6.4. Zadania i problemy

Zadanie 6.1. Narysuj drzewo porządkowania trzech elementów metodą bąbelkową.

Zadanie 6.2. Zbadaj działanie algorytmów porządkowania bąbelkowego i przez wybór na ciągach liczb, które są już uporządkowane od najmniejszej do największej. Posłuż się przy tym programami, które są realizacjami tych algorytmów. Ile porównań jest wykonywanych w każdym z tych algorytmów na ciągu złożonym z 10, 20 i 100 elementów? Uogólnij te odpowiedzi na dowolną liczbę n elementów w ciągu. Możesz ułatwić sobie zadanie, korzystając z aplikacji *Sortowanie* [Oprog], w której jest zliczana liczba operacji, a także można dzięki niej porównywać algorytmy między sobą pod względem liczby wykonywanych działań.

Zadanie 6.3. Wykonaj poprzednie zadanie dla ciągów liczb uporządkowanych w odwrotnej doposzukiwanej kolejności, czyli od największej do najmniejszej

liczby.

Problem 6.1. Liczby będące kwadratami liczb naturalnych, czyli mające postać n^2 , starożytni Grecy nazywali **liczbami kwadratowymi**, gdyż mogą być reprezentowane w postaci kwadratu złożonego z n^2 diamentów. Podziel odpowiednio liniami taki kwadrat i uzasadnij prawdziwość następujących równości:

$$1 + 2 + \dots + (n - 1) + n + (n - 1) + \dots + 2 + 1 = n^2,$$

$$1 + 3 + 5 + \dots + (2n - 1) = n^2.$$

Wskazówka. W przypadku pierwszej równości linie te mogą być odcinkami linii prostych, a w przypadku drugiej — nie muszą.

Problem 6.2. Przypuśćmy, że każdy element danego ciągu ma jedną z dwóch możliwych wartości (np. 0 lub 1). Podaj algorytm porządkowania tego ciągu, którego czas działania jest proporcjonalny do liczby elementów w ciągu.

Zadanie 6.4. Podobnie jak w przykładzie omówionym w tym rozdziale, utwórz słowa (mogą mieć różną długość) składające się z liter, które występują w słowie ALGORYTM, i uporządkuj je algorytmem pozycyjnym. Zauważ, że jednym z tych słów może być anagram tego słowa, który pojawił się już w tej książce i ma bardzo ważne znaczenie w informatyce!

Zadanie 6.5. Dane są dwie daty ery nowożytnej. Podaj algorytm, który posłuży Ci do obliczenia, ile dni dzieli je w kalendarzu. Pamiętaj o latach przestępnych. Zrealizuj swój algorytm w wybranej reprezentacji, a następnie użyj go do obliczenia: ile dni minęło od dnia Twojego urodzenia oraz ile dni minęło od ostatniego przełomu tysiącleci.

Problem 6.3. Podaj inny niż omówiony w tym rozdziale przykład rodzaju elementów, które można porządkować metodą pozycyjną. Opisz szczegółowo algorytm ich porządkowania.

Problem 6.4. Zmodyfikuj procedurę KubelkiLiczb tak, aby otrzymać procedurę KubelkiWyrazow, która jest realizacją algorytmu porządkowania pozycyjnego słów o różnej długości.

Problem 6.5. W porządkowaniu kubełkowym pojedynczych elementów (znaków lub liczb), zamiast odkładania ich do kubełków, można tylko liczyć, ile ich jest. Jest to idea stojąca za metodą **porządkowania przez zliczanie**, która na ogół jest stosowana przy szukaniu lidera lub zwycięzcy wyborów (patrz punkt 5.6.1). Zaprogramuj ten algorytm w wybranym języku programowania. Określ, jaka jest jego złożoność, ale najpierw zastanów się, jakie działania powinny być uwzględnione w złożoności tego algorytmu.

Powinieneś umieć stosować algorytmy porządkowania zbiorów, zawierających dowolną liczbę elementów:

- ▶ algorytm **bąbelkowy**, który działa na zasadzie poprawiania uporządkowania sąsiednich elementów stojących w złej kolejności;
- ▶ algorytm porządkowania **przez wybór**, w którym buduje się właściwy porządek, wybierając do niego kolejne elementy w porządku ich występowania w rozwiązaniu;
- ▶ algorytmy porządkowania **kubelkowego** i **pozycyjnego**, których najważniejszym zastosowaniem jest porządkowanie elementów złożonych z wielu pozycji, które mogą zawierać niejednorodne wartości.

Rozdział 7. Inne algorytmy iteracyjne – schemat Hornera, algorytm Euklidesa, sito Eratostenesa

Tu dowiesz się:

- ▶ ile miejsca w pamięci komputera zajmują liczby;

oraz poznasz:

- ▶ sposób zapisywania liczb w różnych **systemach pozycyjnych**: binarnym, dziesiętnym i szesnastkowym;
- ▶ **schemat Hornera** i jego zastosowania do: szybkiego obliczania wartości wielomianu, obliczania dziesiętnych wartości liczb zapisanych w innych systemach, szybkiego podnoszenia liczb do potęg;
- ▶ słynny **algorytm Euklidesa** i jego zastosowania do: znajdowania największego wspólnego dzielnika liczb, rozwiązywania prostych równań na liczbach całkowitych i wykonywania działań na ułamkach zwykłych;
- ▶ **algorytm rozkładu liczby** na czynniki pierwsze i **sito** do odsiewania liczb złożonych;
- ▶ algorytm stosowany do **obliczeń numerycznych**, który wyznacza rozwiązanie przybliżone **metodą iteracyjną**.

W poprzednich dwóch rozdziałach stosowaliśmy algorytmy iteracyjne, działające na zbiorach liczb lub na zbiorach innych obiektów, którym wprost lub pośrednio można przyporządkować liczby. Będziemy jeszcze wielokrotnie wracać do algorytmów przetwarzających zbiory liczb. W tym rozdziale algorytmy iteracyjne zostaną wykorzystane w rozwiązaniach wybranych zadań obliczeniowych.

7.1. Zapisywanie liczb w systemie binarnym

Zacznijmy rozważania od przypomnienia sobie, co to znaczy, że liczby, którymi się posługujemy na co dzień, są zapisane w **systemie dziesiętnym**.

Ćwiczenie 7.1. Co to znaczy, gdy mówimy, że liczba 73051 jest liczbą dziesiętną? A jeśli byłaby to liczba zapisana w systemie ósemkowym, to która z nich będzie większa? A w systemie szesnastkowym? ■

Jesteśmy tak przyzwyczajeni do automatycznego wykonywania rachunków w systemie dziesiętnym, że niemal nigdy nie korzystamy świadomie z tego, że

poszczególne cyfry liczb w tym systemie spełniają następującą równość (podajemy ją dla przykładowej liczby):

$$7 \cdot 10^4 + 3 \cdot 10^3 + 0 \cdot 10^2 + 5 \cdot 10^1 + 1 \cdot 10^0 = 73051.$$

Liczba 10 w tym zapisie nazywa się **podstawą systemu liczenia**. Jeśli liczba 73051 byłaby zapisana w systemie ósemkowym, co powinniśmy oznaczyć $(73051)_8$, to jej wartość dziesiętna byłaby równa:

$$(73051)_8 = 7 \cdot 8^4 + 3 \cdot 8^3 + 0 \cdot 8^2 + 5 \cdot 8^1 + 1 \cdot 8^0 = 30249 \quad (7.1)$$

W podobny sposób otrzymamy wartość liczby $(73051)_{16}$ i będzie to największa wartość spośród tych trzech. Liczby możemy porównywać dopiero po obliczeniu ich wartości w systemie o tej samej podstawie.

Ogólnie za **podstawę systemu** można wybrać dowolną liczbę naturalną b , nawet większą niż 10 — wtedy cyfry liczby $(a_{n-1} \dots a_1 a_0)_b$ zapisanej w tym systemie należą do zbioru $\{0, 1, \dots, b-1\}$, a jej wartość dziesiętna a jest określona następującą równością:

$$a = a_n b^n + a_{n-1} b^{n-1} + \dots + a_1 b + a_0 \quad (7.2)$$

W dalszej części książki jest mowa jedynie o binarnej reprezentacji liczb naturalnych w komputerach. Korzystając z niej, można utworzyć reprezentacje wszystkich liczb całkowitych (czyli np. reprezentacje liczb typu integer w języku Pascal), jak i liczb typu real (w języku Pascal) i float (w języku Python).

Jeśli $b = 2$, to system nazywa się **binarnym**, a ciąg cyfr $(a_{n-1} \dots a_1 a_0)_2$ ze wzoru (7.2) nazywa się **binarnym rozwinięciem** liczby a lub po prostu **liczbą binarną** (stosowana jest również terminologia **system dwójkowy**). W tym przypadku cyfra a_i nazywa się bitem i może przyjmować wartości 0 lub 1. Liczby binarne stanowią podstawę arytmetyki komputerowej, ponieważ elektroniczne elementy pamięci oraz logiki komputerów mogą się znajdować w dwóch stanach, które są identyfikowane z cyframi 0 i 1.

Interesuje nas teraz algorytm wyznaczania binarnej reprezentacji naturalnej liczby dziesiętnej a . Zgodnie ze wzorem (7.2), liczba a ma następującą postać binarną dla pewnego n :

$$a = a_n 2^n + a_{n-1} 2^{n-1} + \dots + a_1 2 + a_0 \quad (7.3)$$

W skrócie piszemy zwykle $a = (a_{n-1} \dots a_1 a_0)_2$. W jaki sposób można znaleźć kolejne bity tego rozwinięcia? Aby odpowiedzieć na to pytanie, przypomnijmy sobie najpierw, czym w rozwinięciu binarnym różnią się liczby parzyste (czyli podzielne przez 2 z resztą 0) od liczb nieparzystych (czyli podzielnych przez 2 z resztą 1), np. $12 = (1100)_2$ od $13 = (1101)_2$? Widać, że najmniej znaczący bit rozwinięcia binarnego liczby jest równy reszcie z dzielenia tej liczby przez 2. I rzeczywiście, jeśli podzielimy liczbę a opisaną równaniem (7.3) przez 2, to

iloraz jest równy $a_{2n-1}2^{n-1} + a_{2n-2}2^{n-2} + \dots + a_1$, a reszta wynosi a_0 — jest to najmniej znaczący bit w reprezentacji liczby a . Ponieważ iloraz ma również postać równości (7.3), więc następne bity rozwinięcia znajdujemy w podobny sposób. To postępowanie kończymy, gdy iloraz wynosi 0, gdyż wtedy kolejne reszty będą już cały czas równe 0, a zera na początku rozwinięcia binarnego nie mają żadnego znaczenia, podobnie jak na początku jakiegokolwiek liczby. Prześledźmy ten proces na przykładzie przedstawionym na rysunku 7.1.

Dzielenie	Iloraz	Reszta
19 2	9	1
9 2	4	1
4 2	2	0
2 2	1	0
1 2	0	1

Rysunek 7.1. Przykład tworzenia binarnej reprezentacji liczby dziesiętnej 19. Otrzymaliśmy $19 = (10011)_2$

Zauważmy, że w tym algorytmie reprezentacja liczby jest tworzona od końca, czyli od najmniej znaczącego bitu. Należy o tym pamiętać, gdyż prowadzi to czasem do nieporozumień i błędów.

W obliczeniach ilorazu i reszty z dzielenia liczb całkowitych przez siebie posługujemy się dwiema operacjami wykonywanymi na liczbach całkowitych, których wyniki są również liczbami całkowitymi. Dla dwóch liczb całkowitych k i l , wartością $k \bmod l$ jest reszta z dzielenia k przez l , czyli jest to liczba r spełniająca nierówności $0 \leq r < l$. Z kolei, wartością $k \div l$ jest iloraz całkowity z dzielenia k przez l , czyli wynik dzielenia k przez l obcięty do liczby całkowitej. W dalszej części, te dwie operacje są wykonywane przede wszystkim na liczbach naturalnych. W szczególnym przypadku, jeśli $l = 2$, to $k \bmod 2$ ma wartość 0 — gdy k jest liczbą parzystą, i wartość 1 — gdy k jest liczbą nieparzystą. Z kolei jeśli k jest liczbą parzystą, to $k \div 2$ jest równe $k/2$, a jeśli k jest liczbą nieparzystą, to $k \div 2$ ma wartość $(k - 1)/2$.

Algorytm zamiany liczby dziesiętnej na postać binarną

Dane: Dziesiętna liczba naturalna a .

Wynik: Kolejne bity rozwinięcia binarnego liczby a .

Krok 1. Powtarzaj krok 2., dopóki a jest liczbą większą od zera, w przeciwnym razie zakończ algorytm.

Krok 2. {Kolejny od końca bit rozwinięcia danej liczby a jest równy reszcie z

dzielenia a przez 2. Za nową wartość a przyjmujemy iloraz całkowity z dzielenia a przez 2. Matematyczny zapis tych operacji ma następującą postać:}

Za kolejny bit przyjmij: $a \bmod 2$ i przypisz: $a := a \div 2$. ■

Poniżej podajemy opisy algorytmu zamiany dziesiętnej liczby naturalnej na binarną w językach Pascal i Python.



```
procedure System10na2(b:integer; var k:integer; var a:Tablica0n);  
begin  
  k:=-1;  
  repeat  
    k:=k+1;  
    a[k]:=b mod 2;  
    b:=b div 2  
  until b=0  
end; {System10na2}
```



```
def System10na2(n):  
  if n ==0:  
    lista=[0]  
  else:  
    lista=[]  
    while n > 0:  
      lista=[n % 2] + lista  
      n=n // 2
```

Zauważmy w procedurze w języku Pascal, że iteracja w tej procedurze jest zrealizowana za pomocą warunkowej instrukcji iteracyjnej repeat, czyli dla każdej danej liczby b krok iteracyjny jest wykonywany przynajmniej jeden raz. Dzięki temu tę procedurę można stosować również wtedy, gdy b ma wartość 0. Rozwinięcie liczby składa się wówczas z jednego bitu 0. Podobnie działa funkcja w języku Python.

Długość rozwinięcia binarnego liczby

Określimy teraz, ile miejsca, czyli ile bitów, zajmuje zapisanie liczby naturalnej w postaci binarnej — są to nieco trudniejsze rozważania. Odpowiedź na to

pytanie jest jednak bardzo ważna dla osób stosujących komputery, nawet dzisiaj, kiedy można korzystać z niemal nieograniczonej pamięci komputerów i nie trzeba się obawiać, że jej zabraknie.

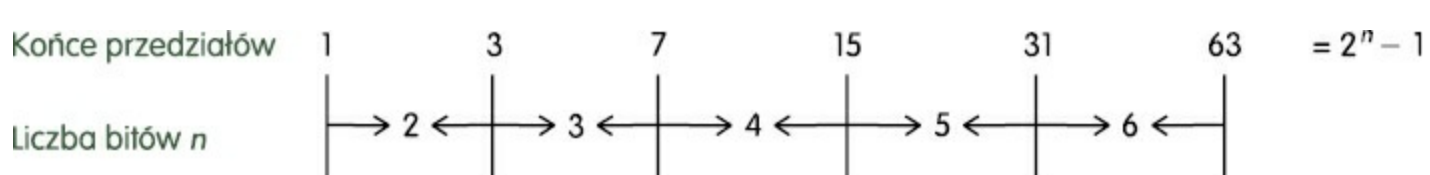
T

Chcemy więc wiedzieć, ile bitów zajmuje liczba naturalna w postaci binarnej. Rozważmy odwrotne pytanie: jaką największą liczbę naturalną można zapisać na n bitach. Taka liczba ma oczywiście wszystkie bity równe 1 w swoim rozwinięciu binarnym ($11\dots11$), a więc jej wartość na podstawie równości (7.3) wynosi:

$$2^{n-1} + 2^{n-2} + \dots + 2 + 1 = 2^n - 1.$$

Aby uzasadnić tę równość, zauważmy że wartość sumy po lewej stronie znaku równości jest liczbą o jeden mniejszą od liczby ($100\dots00$), która w rozwinięciu binarnym składa się z $n + 1$ bitów i ma tylko jeden bit równy 1 — najbardziej znaczący. Tą liczbą jest 2^n . Tę równość można uzasadnić jeszcze inaczej, obliczając sumę ciągu geometrycznego stojącego po jej lewej stronie. Na rysunku 7.2 jest pokazane, ile bitów potrzeba do przedstawienia w postaci binarnej liczb z poszczególnych przedziałów. Z ostatniej równości wynika, że liczba a może być zapisana na n bitach, gdy spełnia nierówność:

$$2^n - 1 \geq a, \text{ czyli } 2^n \geq a + 1.$$



Rysunek 7.2. Liczba bitów potrzebnych do zapamiętania liczb z poszczególnych przedziałów

Czy nie budzi Twojego zdziwienia, że słowo **logarytm** jest anagramem słowa **algorytm**, czyli składa się dokładnie z tych samych liter? To jest jednak przypadek.

Aby była to najmniejsza liczba bitów, wybieramy najmniejsze n spełniające tę nierówność. Używając funkcji logarytmicznej, można dokładnie określić wartość n następująco:

$$n = \lceil \log_2(a + 1) \rceil.$$

Ponieważ wartość logarytmu może nie być liczbą całkowitą, użyliśmy funkcji powała, gdyż liczba bitów w rozwinięciu liczby a jest zawsze liczbą naturalną.

Z tych rozważań należy zapamiętać, że liczba bitów potrzebnych do zapisania w

pamięci komputera liczby naturalnej jest w przybliżeniu równa logarytmowi przy podstawie 2 z wartości tej liczby. Spójrzmy teraz na wykresy funkcji liniowej i logarytmicznej, przedstawione na rysunku 5.4. Widać na nich, że funkcja logarytmiczna rośnie o wiele, wiele wolniej niż funkcja liniowa. Tę obserwację potwierdza porównanie wartości liczb i ich logarytmów w tabeli zamieszczonej tamże.

Polecamy pracę [Log] o znaczeniu logarytmów i funkcji logarytm w algorytmice i w informatyce.

7.2. Schemat Hornera

W rozdziale 2. przedstawiliśmy znacznie prostszy sposób obliczania wartości wielomianów drugiego i trzeciego stopnia niż to wynika ze zwykle stosowanego zapisu tych wielomianów. Przypomnijmy, do jakich postaci doprowadziliśmy te wielomiany:

$$w(x) = ax^2 + bx + c = (ax + b)x + c;$$

$$v(x) = ax^3 + bx^2 + cx + d = ((ax + b)x + c)x + d.$$

W obu przypadkach, liczba dodawań i mnożeń w algorytmach wynikających z tych zapisów jest równa odpowiednio 2 i 3, czyli jest równa stopniowi wielomianu.

Podobnie można przedstawić wielomian stopnia n dla dowolnego $n \geq 0$, który ma następującą ogólną postać:

$$w_n(x) = a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1}x + a_n \quad (7.4)$$

Ćwiczenie 7.2. Zanim przekształcimy ten wielomian, wyznacz, ile mnożeń i dodawań trzeba wykonać, by obliczyć jego wartość, przeprowadzając te działania od prawej do lewej. ■

Przekształcanie wielomianu (7.4) rozpoczynamy od wyłączenia x ze wszystkich wyrazów, w których występuje, po czym otrzymujemy:

$$w_n(x) = (a_0 x^{n-1} + a_1 x^{n-2} + \dots + a_{n-1})x + a_n.$$

Kontynuując to postępowanie dla zagłębiających się wielomianów, uzyskujemy następującą postać wielomianu (7.4):

$$w_n(x) = (\dots((a_0 x + a_1)x + a_2)x + \dots + a_{n-1})x + a_n \quad (7.5)$$

Rozstawienie nawiasów wyznacza następującą kolejność wykonywania działań:

► przyjmij współczynnik przy najwyższej potędze za początkową wartość wielomianu;

► powtarzaj aż do wyczerpania listy współczynników: bieżącą wartość

wielomianu pomnóż przez x i dodaj kolejny współczynnik wielomianu.

Przypuśćmy, że chcemy obliczyć $wn(z)$, czyli wartość wielomianu (7.4) dla wartości argumentu $x = z$ — oznaczmy tę wartość przez y . Wtedy obliczenia przebiegające zgodnie ze wzorem (7.5) można zapisać następująco:

$$y := a_0$$

$$y := yz + a_i \quad i = 1, 2, \dots, n \quad (7.6)$$

W 1819 roku W. G. Horner podał sposób obliczania wartości wielomianu, nazywany dzisiaj jego nazwiskiem, ale 150 lat wcześniej Isaac Newton stosował podobny sposób obliczania wartości wielomianu, który występował w jego rachunkach fizycznych. Te fakty z historii jeszcze raz dowodzą, że wiele algorytmów stosowanych dzisiaj w obliczeniach komputerowych pochodzi z czasów, gdy jeszcze nie było komputerów. Ćwiczenie 7.3 powinno przekonać Cię ponadto, że te algorytmy mogą być również przydatne w obliczeniach za pomocą zwykłych kalkulatorów.

Postać wielomianu (7.5), jak i wynikający z niej sposób obliczania wartości wielomianu (7.6), nazywa się **schematem Hornera**. Ze wzoru (7.6) wynika, że obliczenie wartości wielomianu stopnia n wymaga wykonania n mnożeń i n dodawań.

Ćwiczenie 7.3. Masz do dyspozycji jedynie kalkulator. Oblicz dziesiętną wartość liczby ósemkowej 73051 (zobacz wzór 7.1) w sposób zasugerowany w ćwiczeniu 7.2 oraz posługując się schematem Hornera (7.5). Postaraj się nie używać komórki pamięci kalkulatora. Czy w obu przypadkach jest to możliwe? ■

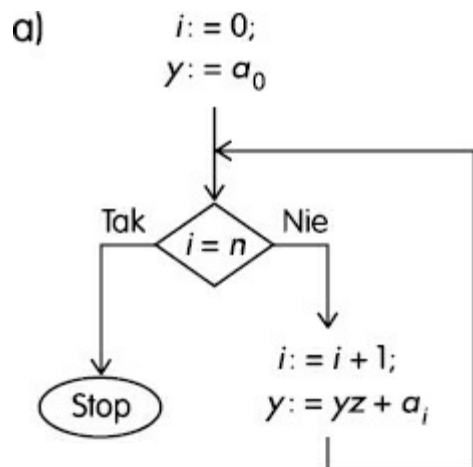
Nie opuszczaj tego ćwiczenia — możesz się nauczyć, jak szybko oblicza się wartość dziesiętną liczb binarnych, a ogólnie — jak szybko oblicza się wartość wielomianu za pomocą prostego kalkulatora.

Na rysunku 7.3 jest przedstawiony opis schematu Hornera (7.6) w postaci schematu blokowego, a dalej — programy w językach Pascal i Python, utworzone dla następującej specyfikacji:

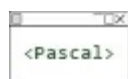
Obliczanie wartości wielomianu

Dane: n, a_0, a_1, \dots, a_n — stopień i współczynniki wielomianu (7.4); z — argument;

Wynik: Wartość wielomianu (7.4) dla argumentu z . ■



Rysunek 7.3. Realizacja schematu Hornera — schemat blokowy
 Programy w językach Pascal i Python realizujące schemat Hornera.



```

function Horner(n:integer; a:Tablica0nr; x:real):real;
  var i:integer; y:real;
begin
  y:=a[0];
  for i:=1 to n do y:=y*x+a[i];
  Horner:=y
end; {Horner}
  
```



```

def Horner(n,lista,x):
  y=lista[0]
  for i in range(1,n,1):
    y=y*x+lista[i]
  return(y)
  
```

A. Borodin udowodnił w 1971 roku, że schemat Hornera jest najszybszym sposobem obliczania wartości wielomianu. Jest to więc kolejny przykład algorytmu optymalnego.

Danymi w tym problemie są wszystkie współczynniki wielomianu, nawet gdy niektóre z nich są równe zeru. Najlepiej to widać w przedstawionych powyżej programach w obu przypadkach obliczenia są wykonywane dla każdego współczynnika wielomianu. To jest koszt, jaki czasem musimy ponosić w automatycznych obliczeniach. Jeśli mielibyśmy uwzględnić specjalne wartości współczynników, należałoby to zrobić w algorytmie — wtedy oczywiście

zburzylibyśmy prostotę i zniszczyli elegancję schematu Hornera.

7.3. Zastosowania schematu Hornera

Na lekcjach matematyki w szkole nie ma zbyt wielu okazji do stosowania schematu Hornera do obliczania wartości wielomianów wyższych stopni, może z wyjątkiem wielomianów stopnia 2. i 3. — w tych przypadkach oszczędność na liczbie mnożeń jest również warta uwzględnienia. W tym punkcie przedstawiamy zastosowania schematu Hornera związane z interpretacją szczególnej postaci wielomianu.

7.3.1. Zamiana liczb z systemu binarnego na dziesiętny

Równość (7.2), będąca zapisem liczby a w systemie pozycyjnym o podstawie b , ma postać wielomianu (7.4), w którym zamiast x występuje zmienna b , a współczynnikami są cyfry rozwinięcia $a_n, a_{n-1}, \dots, a_1, a_0$. Jest to bardzo ważna obserwacja, gdyż dzięki temu w obliczeniach na rozwinięciach liczb w różnych systemach możemy posługiwać się algorytmami, które zostały opracowane specjalnie dla wielomianów. Najprostszym przykładem takiego algorytmu jest właśnie schemat Hornera.

Uwaga: Współczynniki w rozwinięciu (7.2) są odwrotnie ponumerowane niż w postaci wielomianu (7.4) — pamiętaj o tym.

Po wykonaniu ćwiczenia 7.3 wiesz już, że schemat Hornera może być stosowany do obliczania wartości dziesiętnej liczby a , jeśli dane jest jej rozwinięcie przy dowolnej podstawie b . W szczególnym przypadku można korzystać ze schematu Hornera przy zamianie liczb binarnych na dziesiętne. Algorytm jest identyczny ze schematem Hornera, należy tylko pamiętać, by „współczynniki” wielomianu (tj. kolejne cyfry rozwinięcia) były podawane od najbardziej znaczącej, czyli w odwrotnej kolejności niż są obliczane w algorytmie tworzenia postaci binarnej. Można jednak tę kolejność cyfr łatwo odwrócić (zobacz problem 7.2).

7.3.2. Szybkie podnoszenie do potęgi

W zadaniu 2.1 odpowiadałeś na pytanie, jak najszybciej można obliczyć szóstą potęgę danej liczby x . Wykonywanie kolejnych mnożeń $x \cdot x \cdot x \cdot x \cdot x \cdot x$ to pięć działań. By zmniejszyć ich liczbę, można skorzystać z równości $6 = 2 \cdot 3$ i najpierw podnieść x do kwadratu, a następnie wynik podnieść do trzeciej potęgi, gdyż $x^6 = (x^2)^3$. Zatem zamiast pięciu mnożeń, wystarczy wykonać tylko trzy. A jeśli mielibyśmy podnieść x do potęgi 16? Wtedy wystarczy wykonać tylko cztery mnożenia zamiast piętnastu — szalona oszczędność.

W tym samym zadaniu znajduje się również sugestia dotycząca użycia binarnej postaci wykładnika potęgi $6 = (110)_2$. Zatem jeśli dodatkowo skorzystamy ze schematu Hornera, to otrzymamy:

$$x^6 = x^{(1 \cdot 2 + 1) \cdot 2} = (x^2 x)^2.$$

Ponieważ każdą liczbę naturalną można przedstawić w postaci binarnej, ten sposób podnoszenia do potęgi wydaje się być uniwersalny i może być użyty dla dowolnej liczby naturalnej m . Dla uproszczenia początkowych rozważań przyjmijmy, że wykładnik m jest pamiętany na trzech bitach, a więc ma postać $m = (m_2 2 + m_1) 2 + m_0$ oraz $m_2 = 1$. Wówczas otrzymamy następującą równość, która wynika jedynie z zastosowania zasad potęgowania:

$$x^m = x^{(m_2 2 + m_1) 2 + m_0} = \left((x^{m_2})^2 x^{m_1} \right)^2 x^{m_0} = (x^2 x^{m_1})^2 x^{m_0}$$

Na tej podstawie można wyprowadzić następującą zależność między postacią wykładnika w układzie binarnym a działaniami wykonywanymi przy obliczaniu potęgi:

- najbardziej znaczący bit w rozwinięciu wykładnika (który jest zawsze równy 1) odpowiada rozpoczęciu obliczeń od przyjęcia liczby za początkową wartość potęgi;
- każda następna pozycja w rozwinięciu odpowiada podniesieniu częściowego wyniku do kwadratu i ewentualnie pomnożeniu przez x , jeśli bit rozwinięcia na tej pozycji jest równy 1.

Opisany obok sposób podnoszenia do potęgi znany był w Indiach 200 lat p.n.e., a w X wieku pojawił się w Damaszku.

Tę zasadę można zobrazować lepiej. Niech P oznacza podniesienie do kwadratu, a X — pomnożenie przez x . Wówczas opisany sposób podnoszenia x do potęgi m można przedstawić następująco: Utwórz najpierw ciąg liter P i X , przyporządkowując każdemu bitowi w rozwinięciu binarnym wykładnika m (z wyjątkiem najbardziej znaczącego) symbole: PX — gdy jest on równy 1, i P — gdy jest równy 0. Na przykład dla $m = 6 = (110)_2$ otrzymamy ciąg: PXP .

Następnie za początkową wartość potęgi przyjmij x i wykonuj działania zgodnie z tym ciągiem (zaczynając od lewej strony), pamiętając o znaczeniu liter P i X . Ten algorytm można nazwać algorytmem binarnym „od lewej do prawej”, gdyż wykonuje działania zgodnie z taką właśnie kolejnością bitów w rozwinięciu wykładnika.

Binarny algorytm potęgowania od lewej do prawej

Dane: Liczba naturalna m w postaci binarnej ($m_{n-1} m_{n-2} \dots m_1 m_0$) i dowolna liczba x .

Wynik: Wartość potęgi x^m .

Krok 1. {Ustalenie początkowej wartości potęgi.} $y := x$.

Krok 2. Dla $i = n - 1, n - 2, \dots, 1, 0$ wykonaj: jeśli $m_i = 1$, to $y := y \cdot y \cdot x$, w przeciwnym razie $y := y \cdot y$. ■

Przedstawmy kolejność działań w tym algorytmie dla wykładnika $m = 22$. Mamy $22 = (10110)_2$. Zatem kolejnymi wartościami zmiennej y w tym algorytmie są: x , xx , $x^2x^2x = x^5$, $x^5x^5x = x^{11}$, $x^{11}x^{11} = x^{22}$. Taką samą kolejność działań otrzymamy, tworząc ciąg liter P i X, który w tym przykładzie ma postać PPXPXP.

Obliczymy, ile mnożeń wykonujemy w tym algorytmie. Wszystkie te działania są wykonywane w kroku 2., dla kolejnych bitów w binarnej reprezentacji wykładnika, z wyjątkiem pierwszego, najbardziej znaczącego bitu. Dla każdej pozycji, bez względu na wartość bitu, jest wykonywane mnożenie $y \cdot y$, oraz jeśli bit wynosi 1, to również dodatkowe mnożenie przez x . Zatem liczba mnożeń w całym algorytmie jest równa liczbie wszystkich bitów w binarnej reprezentacji wykładnika, minus jeden, plus liczba jedynek w tej reprezentacji, również minus jeden, gdyż najbardziej znaczącej jedynce odpowiada początkowa wartość $y = x$ i nie jest wykonywane żadne mnożenie (zobacz problem 7.3).

Algorytm potęgowania od lewej do prawej ma pewną wadę. Jak pamiętamy, rozwinięcie binarne liczby jest naturalnie tworzone od najmniej znaczącego bitu, czyli od prawej do lewej. Można odeprzeć ten argument stwierdzeniem, że przecież liczby naturalne są pamiętane w komputerze w postaci binarnej, więc binarna reprezentacja wykładnika jest dana. Jednak, aby z niej skorzystać, musimy i tak dotrzeć do poszczególnych jej bitów. Najłatwiej to zrobić, stosując algorytm, który tworzy taką właśnie reprezentację bit po bicie, jak opisany w punkcie 7.1.

Ćwiczenie 7.4. Utwórz schemat blokowy binarnego algorytmu potęgowania od lewej do prawej przyjmując, że binarna reprezentacja wykładnika jest dana w kolejności od najbardziej znaczącego bitu. ■

Opiszemy teraz algorytm potęgowania, który również korzysta z binarnej reprezentacji wykładnika, ale tworzy ją w trakcie obliczania potęgi, od prawej do lewej. W tym przypadku postać binarna nie jest zamieniana na schemat Hornera — kolejnymi wykładnikami czynników są potęgi liczby 2 występujące w rozwinięciu binarnym wykładnika. Dla $m = 6$ potęga jest więc obliczana według wzoru:

$$x^6 = x^{2^2+2^1} = x^4 x^2$$

Algorytm, który podajemy obok, jest również bardzo stary — znany był w XV wieku, a już egipscy matematycy 1800 lat p.n.e. w podobny sposób znajdowali

wartość iloczynu nx .

Binarny algorytm potęgowania od prawej do lewej

Dane: Liczba naturalna m i dowolna liczba x .

Wynik: Wartość potęgi x^m .

Krok 1. {Ustalenie początkowych wartości potęgi i zmiennych.}

$y := 1, z := x, l := m.$

Krok 2. $l := l \text{ div } 2$ i jednocześnie sprawdź, jakiej parzystości było l przed podzieleniem. Jeśli l było parzyste, to przejdź do kroku 5.

Krok 3. $y := y \cdot z.$

Krok 4. Jeśli $l = 0$, to zakończ algorytm — wynikiem jest bieżąca wartość y .

Krok 5. $z := z \cdot z$. Wróć do kroku 2. ■

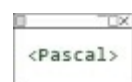
Na rysunku 7.4 są przedstawione wyniki wykonania poszczególnych kroków tego algorytmu dla $m = 6$.

	l	z	y
Po kroku 1:	6	x	1
Po kroku 5:	3	x^2	1
Po krokach 3 i 5:	1	x^4	x^2
Po krokach 3 i 4:	0		x^6

Rysunek 7.4. Przykład działania algorytmu potęgowania od prawej do lewej dla wykładnika $m = 6$

Ćwiczenie 7.5. Określ kolejne wartości zmiennej y w algorytmie potęgowania od prawej do lewej dla $m = 22$ i porównaj z wykonaniem algorytmu potęgowania od lewej do prawej dla tej samej wartości m . ■

W algorytmie potęgowania od prawej do lewej wykonujemy o jedno więcej mnożenie niż w poprzednim algorytmie i wykorzystujemy o jedną zmienną pomocniczą z więcej — są w niej pamiętane wartości potęgi liczby x dla wykładników będących kolejnymi potęgami liczby 2.



```
function Potega0dPrDoLe(m:integer; x:real):real;  
  var j,l:integer;  
      y,z:real;  
  
begin
```

```
y:=1; z:=x; l:=m;
```

```
repeat
```

```
  j:=l mod 2;
```

```
  l:=l div 2;
```

```
  if j=1 then y:=y*z;
```

```
  z:=z*z
```

```
until l=0;
```

```
PotegaOdPrawejDoLewej:=y
```

```
end; {PotegaOdPrDoLe}
```



```
def PotegaOdPrDoLe(m,x):
```

```
    y=1
```

```
    z=x
```

```
    l=m
```

```
    while l > 0:
```

```
        j=l % 2
```

```
        l=l // 2
```

```
        if j == 1:
```

```
            y=y*z
```

```
        z=z*z
```

```
    return y ■
```

Ćwiczenie 7.6. Policz dokładnie, ile działań wykonują obie implementacje algorytmu potęgowania od prawej do lewej, w języku Pascal i w języku Python. Możesz w tym celu użyć w obu implementacjach dodatkowych zmiennych, w których będą zliczane poszczególne operacje. ■

Problem szybkiego potęgowania liczb zajmuje matematyków od dawna i nadal nie jest znany ogólny algorytm, który dla każdego wykładnika wykonywałby najmniejszą liczbę działań. Dwa inne sposoby potęgowania są zasugerowane w problemach 7.4 i 13.27.

W tym miejscu może pojawić się pytanie, czy metoda binarna jest najszybszym sposobem potęgowania. Przykładu na to, że tak niestety nie jest, nie trzeba szukać daleko. Dla $m = 15$ pierwszy z algorytmów binarnych wykonuje sześć mnożeń, a wystarczy wykonać pięć — najpierw obliczamy $y = x^3$, wykonując dwa mnożenia, a następnie obliczamy piątą potęgę y^5 za pomocą trzech dodatkowych mnożeń.



Szczegółowe omówienie teorii i algorytmów potęgowania znajdziesz w książce [Knuth-2, punkt 4.6.3]. ■

7.4. Algorytm Euklidesa

Przedstawiamy w tym punkcie algorytm, który został odkryty jako jeden z najwcześniejszych, bo jeszcze w starożytności, przez Euklidesa.

Danymi są dwie nieujemne liczby całkowite m i n . Liczba k jest **największym wspólnym dzielnikiem** m i n , jeśli dzieli m oraz n i jest największą liczbą o tej własności — oznaczamy ją przez $\text{NWD}(m, n)$. **Algorytm Euklidesa** znajduje największy wspólny dzielnik dwóch liczb. Jest to jeden z algorytmów najczęściej przytaczanych w książkach informatycznych i matematycznych. Powodem tego jest nie tylko szacunek dla jego „wieku”, ale to, że po pierwsze — ma wiele zastosowań w rozwiązywaniu problemów rachunkowych (zobacz punkt 7.5), a po drugie — można na jego przykładzie zilustrować wiele podstawowych pojęć i własności informatycznych (zobacz algorytm rekurencyjny w punkcie 8.1.3 i dowodzenie poprawności algorytmów w problemach 12.1 i 12.2).

Ćwiczenie 7.7. Pierwszą metodą, jaka może przyjść na myśl, gdy chcemy znaleźć największy wspólny dzielnik dwóch danych liczb m i n , jest sprawdzanie podzielności tych liczb przez kolejne liczby naturalne, począwszy od 2, a skończywszy na mniejszej z liczb m lub n . Ile dzieleni należy wykonać, aby obliczyć tą metodą $\text{NWD}(24, 48)$ oraz $\text{NWD}(46, 48)$? A dla dowolnych m i n ? Opisz tę metodę w postaci algorytmu w wybranej przez siebie reprezentacji. ■

Nie możesz być chyba zadowolony z efektywności tego algorytmu, zwłaszcza dla przykładowych danych liczbowych — w pierwszym przypadku, gdy jedna z liczb dzieli drugą, od razu widać, że ta mniejsza jest poszukiwanym największym wspólnym dzielnikiem obu liczb. A gdy żadna z liczb nie dzieli drugiej, tak jak w drugim przypadku?

Założmy, że $n \geq m$. Gdy podzielimy n przez m , otrzymamy następującą równość:
$$n = qm + r, \text{ gdzie } 0 \leq r < m \quad (7.7)$$

Algorytm, który opisujemy w tym punkcie, Euklides zamieścił w swoim wiekopomnym dziele *Elementy*, w którym m.in. położył podwaliny pod geometrię na płaszczyźnie, nazywaną również jego nazwiskiem. Było to około 300 roku p.n.e., a więc na długo przed pojawieniem się określenia „algorytm”. Przypuszcza się, że ten algorytm był znany jeszcze wcześniej. Wiele innych przepisów rachunkowych stosowano już w Babilonie blisko 1800 roku p.n.e.

Jednak nie nadaje się im takiego znaczenia, jak algorytmowi Euklidesa, gdyż nie zawierają w sobie iteracji i wiele z nich zostało wypartych przez współczesne metody.

Wielkości q i r są odpowiednio **ilorazem** i **resztą** z dzielenia n przez m . Po podstawieniu danych z ćwiczenia 7.7 do równości (7.7) otrzymamy: $48 = 2 \cdot 24$ i $48 = 1 \cdot 46 + 2$. Wynikają stąd następujące wnioski:

- ▶ jeśli $r = 0$, to $\text{NWD}(m, n) = m$, czyli jeśli jedna z liczb dzieli drugą bez reszty, to mniejsza z tych liczb jest ich największym wspólnym dzielnikiem;
- ▶ jeśli $r \neq 0$, to równość (7.7) można zapisać w postaci $r = n - qm$; stąd wynika, że każda liczba dzieląca n i m dzieli całe wyrażenie po prawej stronie tej równości, a więc dzieli również r ; zatem największy wspólny dzielnik m i n dzieli również resztę r .

Te dwa wnioski można zapisać w postaci następującej równości:

$$\text{NWD}(m, n) = \text{NWD}(r, m),$$

w której przyjęliśmy, że $\text{NWD}(0, m) = m$, gdyż 0 jest podzielne przez każdą liczbę różną od zera.

Z równości (7.7) mamy $r < m$, a zatem pierwszy element w parze argumentów funkcji NWD maleje. Ponieważ w obliczeniach występują tylko liczby naturalne, ten malejący ciąg reszt musi być skończony, czyli ostatecznie reszta osiąga wartość zero. Z pierwszej zaś części powyższego wniosku wynika, że w takim przypadku mniejsza z liczb m i n jest szukanym największym wspólnym dzielnikiem tych liczb, czyli również liczb, z którymi rozpoczęliśmy obliczenia. Na przykład dla drugiej pary liczb z ćwiczenia 7.7 otrzymujemy:

$$48 = 1 \cdot 46 + 2,$$

$$46 = 23 \cdot 2 + 0,$$

$$\text{czyli } \text{NDW}(46, 48) = \text{NWD}(2, 46) = 2.$$

Podsumujmy to rozumowanie. Zależność (7.7) zapewnia, że możemy generować pary liczb o tym samym największym wspólnym dzielniku, elementy tych par tworzą malejący ciąg liczb naturalnych, a więc ten ciąg jest skończony i na jego końcu otrzymujemy szukany dzielnik. Jest to skrót rozumowania uzasadniającego formalnie poprawność algorytmu Euklidesa (szczegóły pozostawiamy do samodzielnego uzupełnienia, zobacz problemy 12.1 i 12.2).

Jesteśmy już przygotowani do przedstawienia ścisłego opisu algorytmu Euklidesa. Zauważmy jeszcze, że iloraz q z równości (7.7) nie występuje w następnych iteracjach tej równości, zatem nie trzeba go wyznaczać.

Algorytm Euklidesa (z dzieleniem) wyznaczania NWD

Dane: Dwie liczby naturalne m i n , $m \leq n$.

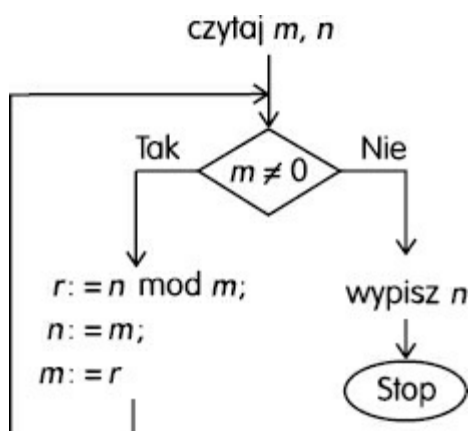
Wynik: $\text{NWD}(m, n)$ — największy wspólny dzielnik m i n .

Krok 1. Jeśli $m = 0$, to n jest szukany dzielnikiem. Zakończ algorytm.

Krok 2. $r := n \bmod m, n := m, m := r$. Wróć do kroku 1. ■

Resztę r w kroku 2. można znaleźć przez kolejne odejmowanie m od n , co jest źródłem dość popularnej realizacji algorytmu Euklidesa z odejmowaniem. Porównajmy obie wersje algorytmu na parze danych z ćwiczenia 7.7. Powyżej wykonaliśmy już obliczenia pierwszym z algorytmów: kończy on działanie po dwóch iteracjach kroku 2. Jeśli zastosujemy odejmowanie zamiast obliczania wartości funkcji mod, w pierwszym kroku — wynik $48 \bmod 46$ otrzymamy po wykonaniu jednego odejmowania, 46 od 48. Natomiast w drugiej iteracji kroku 2., wynik $46 \bmod 2$ otrzymamy po wykonaniu dwudziestu trzech odejmowań. Liczba działań w tym drugim algorytmie, zwłaszcza odejmowań, może więc być znacznie większa niż liczba obliczeń reszty w pierwszym algorytmie.

Na rysunku 7.5 jest przedstawiony klasyczny schemat blokowy algorytmu Euklidesa z dzieleniem, a poniżej jego implementacje w obu językach programowania.



Rysunek 7.5. Algorytm Euklidesa — schemat blokowy



```
function EuklidDziel(m,n:integer):integer;
begin
  while (m>0) and (n>0) do
    if m > n then m:=m mod n
    else n:=n mod m;
  EuklidDziel:=m+n
end; {EuklidDziel}
```



```
def EuklidDziel(m,n):
    while m > 0 and n > 0:
        if m > n:
            m=m % n
        else:
            n=n % m
    return m + n
```

Ciekawym zadaniem jest określenie zależności między liczbą iteracji kroku 2. w algorytmie Euklidesa a wartościami liczb m i n . Nie podamy tutaj pełnej odpowiedzi na to pytanie, a jedynie podpowiedzi w postaci przykładów liczb, dla których ten algorytm wykonuje kilka lub bardzo dużo iteracji.

Ćwiczenie 7.8. Posługując się zależnością (7.7), podaj, dla jakich par liczb m i n algorytm Euklidesa z dzieleniem wykonuje dokładnie jedną iterację kroku 2., a dla jakich — dokładnie dwie iteracje? ■

A teraz trochę trudniejsze zadanie.

Ciąg reszt, który pojawia się w rozwiązaniu zadania 7.1, wystąpi jeszcze kilkakrotnie w tej książce. Ma on duże znaczenie w informatyce.

Zadanie 7.1. Zastosuj algorytm Euklidesa do liczb 34 i 55. Co ciekawego możesz powiedzieć o działaniu algorytmu w tym przypadku — ile wynoszą kolejne ilorazy? Wypisz od końca ciąg reszt tworzonych w tym przypadku. Jak można inaczej zdefiniować ten ciąg? Podaj inną parę liczb, dla której algorytm Euklidesa działa podobnie. ■



Algorytm Euklidesa ma bardzo bogatą literaturę. Najobszerniej jest omówiony w książce [Knuth-2]. Wiele jego elementarnych własności opisano w książkach [EI-I, EI-II] oraz [Graham]. ■

7.5. Zastosowania algorytmu Euklidesa

7.5.1. Przelewanie wody

Znana jest następująca łamigłówka.

Ćwiczenie 7.9. Dysponujesz: dwoma czerpakami o pojemnościach 4 i 6 litrów, pustym pojemnikiem o nieograniczonej objętości i nieograniczoną ilością wody (np. z kranu). Podaj sposób napełnienia pojemnika 15 litrami wody, przy czym

wodę możesz wlewać do pojemnika lub wylewać z niego tylko pełnymi czerpakami. ■

Liczby 4, 6 i 15 w tym ćwiczeniu można zmieniać — oznaczmy je odpowiednio przez m , n i k . Zatem, dla jakich wartości tych trzech wielkości ćwiczenie 7.9 ma pozytywne rozwiązanie? Przypuśćmy, że takie rozwiązanie istnieje. Oznaczmy przez x i y liczbę „ruchów” czerpakami o pojemnościach m i n , odpowiednio. Tutaj, aby wyznaczyć liczbę „ruchów” naczyniem o wybranej pojemności, obliczamy, ile razy wlewano nim wodę, a ile razy wylewano — i pozostawiamy z odpowiednim znakiem różnicę między większą liczbą a mniejszą. Na przykład, jeśli czerpaka użyto 5 razy, by wlać wodę do pojemnika, a 3 razy, by wylać, to przyjmujemy, że ten czerpak był użyty 2 razy. Jeśli natomiast 3 razy nim wlewano, a 5 razy wylewano, to przyjmujemy, że był użyty -2 razy. Zatem te liczby x i y wraz z pojemnościami czerpaków określają ilość wody w pojemniku, która wynosi $mx + ny$. Aby więc ćwiczenie 7.9 miało pozytywne rozwiązanie, musi być spełnione równanie:

$$mx + ny = k \quad (7.8)$$

dla pewnych liczb całkowitych x i y . Dla jakich liczb m , n i k istnieje rozwiązanie x i y równania (7.8)? Czy dla danych podanych w ćwiczeniu równanie $4x + 6y = 15$ ma rozwiązanie? Prosty argument przekonuje, że nie ma rozwiązania — po lewej stronie bowiem mamy liczbę parzystą, bez względu na wartości x i y , a po prawej — liczbę nieparzystą. A jeśli równanie będzie miało postać $4x + 8y = 10$? To również nie ma rozwiązania, chociaż obie strony są liczbami parzystymi, bo wyrażenie po lewej stronie jest podzielne przez 4, a po prawej — nie jest. Te ostatnią obserwację możemy uogólnić następująco: liczba k w równaniu (7.8) musi być podzielna przez $\text{NWD}(m, n)$, ponieważ lewa strona tej równości jest podzielna przez $\text{NWD}(m, n)$, gdyż każda z liczb m i n dzieli się przez $\text{NWD}(m, n)$.

Stąd otrzymujemy następujący wniosek: równanie (7.8) dla danych m , n i k ma rozwiązanie x i y , gdy k jest równe $\text{NWD}(m, n)$. Wynika stąd także, że k może być również wielokrotnością $\text{NWD}(m, n)$, gdyż wtedy całą operację przelewania wody trzeba powtórzyć tyle razy, ile wynosi ta wielokrotność.

Pozostaje podać teraz sposób znajdowania x i y , gdy dane są m i n (zakładamy, że $k = \text{NWD}(m, n)$). I tutaj wystarczy zastosować algorytm Euklidesa, ale... od tyłu. Pamiętamy, że w algorytmie Euklidesa obliczenia rozpoczynamy od równości (7.7), w której występują m i n , a kończymy równością, w której reszta jest równa 0 i dzielnik jest szukanym największym wspólnym dzielnikiem m i n . Teraz, aby znaleźć x i y , rozpoczynamy od tej ostatniej równości i kończymy na tej, od której rozpoczęliśmy obliczenia, czyli na (7.7). Zilustrujmy to przykładem dla $m = 12$ i $n = 21$, chociaż trudno byłoby operować tak dużymi czerpakami, zwłaszcza pełnymi wody. Stosując równość (7.7), znajdujemy najpierw $\text{NWD}(12,$

21):

$$21 = 1 \cdot 12 + 9,$$

$$12 = 1 \cdot 9 + 3,$$

$$9 = 3 \cdot 3 + 0.$$

Stąd $\text{NWD}(12, 21) = 3$. Aby otrzymać równość w postaci (7.8), drugie z powyższych równań zapisujemy następująco: $12 - 1 \cdot 9 = 3 = \text{NWD}(12, 21)$, natomiast z pierwszego wyznaczamy 9 (czyli $9 = 21 - 1 \cdot 12$) i wstawiamy do drugiego równania. W wyniku otrzymujemy:

$$12 - 1 \cdot (21 - 1 \cdot 12) = 3, \text{ czyli } 2 \cdot 12 - 1 \cdot 21 = 3, \text{ a więc } 2m - n = 3.$$

Zatem w tym przykładzie należy najpierw wlać do pojemnika dwa pełne wody mniejsze czepaki i następnie wylać z niego cały drugi czepak — w pojemniku pozostaną 3 litry wody.

Dla wytrwałych Czytelników — wyprowadzimy teraz algorytm, który dla danych liczb naturalnych m i n w jednym przebiegu jednocześnie oblicza $\text{NWD}(m, n)$ i znajduje rozwiązanie równania (7.8), w którym $k = \text{NWD}(m, n)$.

T

Zapiszmy w postaci iteracji kolejne kroki algorytmu Euklidesa:

$$a_0 = q_1 a_1 + a_2$$

$$a_1 = q_2 a_2 + a_3$$

$$\vdots$$

$$a_{l-1} = q_l a_l + a_{l+1}$$

gdzie dla ujednolicenia przyjęliśmy, że $a_0 = n$, $a_1 = m$ oraz $a_{l+1} = 0$, czyli $a_l = \text{NWD}(m, n)$. Szukamy rozwiązania równania $\text{NWD}(m, n) = mx + ny$. W tym celu algorytm będzie tworzył również dwa ciągi liczb x_0, x_1, \dots, x_l oraz y_0, y_1, \dots, y_l spełniające równość:

$$a_i = mx_i + ny_i \text{ dla } i = 0, 1, 2, \dots, l \quad (7.9)$$

czyli dla $i = l$ otrzymamy $a_l = \text{NWD}(m, n) = mx_l + ny_l$. Zatem po zakończeniu algorytmu końcowe wartości elementów ciągów a_i , x_i oraz y_i będą stanowić rozwiązanie równania (7.8).

Określimy teraz sposób wyznaczania elementów ciągów x_i oraz y_i . Z dwóch pierwszych iteracji algorytmu Euklidesa otrzymujemy ich początkowe wartości:

$$n = a_0 = mx_0 + ny_0, \text{ czyli } x_0 = 0, y_0 = 1,$$

$m = a_1 = mx_1 + ny_1$, czyli $x_1 = 1, y_1 = 0$.

Natomiast gdy skorzystamy z równości w algorytmie Euklidesa, $a_{i+1} = a_{i-1} - q_i a_i$, i wstawimy do niej wartości a_{i-1} oraz a_i , wynikające z zależności (7.9), wówczas otrzymamy:

$$\begin{aligned} a_{i+1} &= a_{i-1} - q_i a_i = mx_{i-1} + ny_{i-1} - q_i(mx_i + ny_i) = \\ &= m(x_{i-1} - q_i x_i) + n(y_{i-1} - q_i y_i). \end{aligned}$$

Jeśli teraz porównamy to równanie z zależnością (7.9) dla $i+1$, to stwierdzimy, że muszą być spełnione następujące równości:

$$x_{i+1} = x_{i-1} - q_i x_i \text{ oraz } y_{i+1} = y_{i-1} - q_i y_i.$$

To kończy definiowanie ciągów a_i , x_i oraz y_i . W tabeli 7.1 są przedstawione wartości kolejnych elementów tych ciągów wyznaczone dla danych z naszego przykładu: $m = 12$ i $n = 21$.

Tabela 7.1. Wartości elementów ciągów a_i , x_i oraz y_i wyznaczone dla $m = 12$ i $n = 21$. Zieloną pogrubioną czcionką zaznaczono rozwiązanie

I a_i q_i x_i y_i

0 21 0 1

1 12 1 0

2 9 1 -1

3 3 3 2 -1

4 0

Oto opis algorytmu wyznaczania rozwiązania równania (7.8) i jednocześnie obliczania NWD. Ponieważ w kolejnych iteracjach algorytmu występują odwołania jedynie do dwóch poprzednich wartości elementów ciągów a_i , x_i oraz y_i , w algorytmie zrezygnowaliśmy ze zmiennych z indeksami — nową wartość zmiennej oznaczamy tą samą literą opatrzoną primem.

Algorytm Euklidesa — wersja rozszerzona

Dane: Dwie liczby naturalne m i n , $m \leq n$.

Wyniki: Największy wspólny dzielnik m i n , $\text{NWD}(m, n)$ oraz rozwiązanie x i y równania (7.8) dla $k = \text{NWD}(m, n)$.

Krok 1. {Przypisanie wartości początkowych.} $a := n$; $a' := m$; $x := 0$; $x' :=$

$1; y := 1; y' := 0.$

Krok 2. Jeśli $a' = 0$, to $a = \text{NWD}(m, n)$ i x oraz y stanowią rozwiązanie równania (7.8) dla $k = \text{NWD}(m, n)$ — zakończ algorytm.

Krok 3. Wykonaj przypisania {Zauważ, że uaktualniając wartości, musimy się posłużyć pomocniczą zmienną z }:

$$\begin{aligned} q &:= a \operatorname{div} a'; \\ z &:= a'; a' := a - qa'; a := z; \\ z &:= x'; x' := x - qx'; x := z; \\ z &:= y'; y' := y - qy'; y := z. \end{aligned}$$

Krok 4. Wróć do kroku 2. ■

Zapisanie tego algorytmu w wybranej reprezentacji pozostawiamy do samodzielnego wykonania.

Łatwa i przyjemna zabawa w przelewanie wody zamieniła się w dość żmudne rachunki, aby wyprowadzić i opisać ogólną postać algorytmu, odpowiednią dla naczyń o dowolnych pojemnościach. Równanie (7.8) jest szczególnym przypadkiem tzw. **równania diofantycznego**, w którym współczynniki są liczbami całkowitymi i rozwiązanie również składa się z liczb całkowitych. Na ogół sposoby rozwiązywania takich równań, gdy istnieją, są bardzo złożone.

7.5.2. Operacje na ułamkach zwykłych

Znajdowanie największego wspólnego dzielnika dwóch liczb (NWD) ma zastosowanie w obliczeniach, w których posługujemy się ułamkami zwykłymi.

Liczbę mającą postać $\frac{p}{q}$ nazywamy **ułamkiem zwykłym**, gdy p i q są liczbami

względnie pierwszymi (tj. gdy ułamek nie można skrócić) oraz $q > 0$. (Dalej w tekście ten ułamek zapisujemy w postaci p/q). Dla uproszczenia rozważań zakładamy, że $p \geq 0$. Jeśli ułamek nie jest zwykły, np. $12/16$, to aby sprowadzić go do postaci zwykłej, należy licznik i mianownik podzielić przez ich największy wspólny dzielnik. W ten sposób w działaniach na ułamkach zwykłych pojawia się operacja znajdowania największego wspólnego dzielnika, stąd też m.in. wynika jej duże znaczenie w różnych obliczeniach.

Przypomnijmy podstawowe działania arytmetyczne na ułamkach zwykłych:

$$\frac{p}{q} \pm \frac{p'}{q'} = \frac{pq' \pm p'q}{qq'},$$

$$\frac{p}{q} \cdot \frac{p'}{q'} = \frac{pp'}{qq'},$$

$$\frac{p}{q} \bigg/ \frac{p'}{q'} = \frac{pq'}{qp'}.$$

Aby wynik tych działań był również ułamkiem zwykłym, należy skrócić ułamki występujące po prawej stronie tych tożsamości, gdy jest to możliwe. Jeśli ułamek może być skrócony, to oczywiście jego licznik i mianownik maleją. W podanych dalej algorytmach wykonywania podstawowych operacji na ułamkach skracanie nie jest odkładane na koniec obliczeń, dzięki temu nie pojawiają się w nich niepotrzebnie zbyt duże liczby. Ma to znaczenie w obliczeniach komputerowych na liczbach całkowitych, gdy zakres tych liczb jest niewielki.

Zacznijmy od prostszego zadania. Za wspólny mianownik w sumie lub różnicy ułamków zwykłych obieramy najmniejszą wspólną wielokrotność mianowników w dodawanych ułamkach. **Najmniejszą wspólną wielokrotnością** liczb naturalnych m i n jest najmniejsza liczba naturalna, która dzieli się przez m i n — oznaczamy ją przez $NWW(m, n)$. Prawdziwy jest następujący wzór:

$$NWW(m, n) = \frac{mn}{NWD(m, n)} \quad (7.10)$$

Wynika on z prostej obserwacji, że każdy czynnik z $NWD(m, n)$ występuje w iloczynie mn dwukrotnie: raz w m , a raz w n . Zatem wykonanie dzielenia po prawej stronie eliminuje jednokrotne wystąpienia czynników, które są w obu liczbach m i n . Licznik we wzorze (7.10) może być jednak dużą liczbą, dlatego NWW dwóch liczb wyznaczamy według następującej równości:

$$NWW(m, n) = m \frac{n}{NWD(m, n)}$$

Zatem najpierw jest obliczana wartość $NWD(m, n)$, przez którą jest dzielona większa z liczb m lub n , i na końcu jest wykonywane mnożenie przez drugą z liczb m lub n . Otrzymujemy stąd następujący algorytm:

Algorytm obliczania najmniejszej wspólnej wielokrotności dwóch liczb (NWW)

Dane: Dwie liczby naturalne m i n , $m \leq n$.

Wynik: $\text{NWW}(m, n)$ — najmniejsza wspólna wielokrotność m i n .

Krok 1. Oblicz $\text{NWD}(m, n)$. {Zastosuj w tym celu algorytm Euklidesa.}

Krok 2. $\text{NWW}(m, n)$ jest równa $m(n \text{ div } \text{NWD}(m, n))$. ■

Pokażemy teraz, jak w działaniach na ułamkach zwykłych można unikać pojawiania się dużych liczb.

Dodawanie i odejmowanie. Podamy jedynie wzór dla dodawania — dla odejmowania jest podobny. Obliczamy najpierw $r = \text{NWD}(q, q')$. Jeśli $r = 1$, to wyniku dodawania nie można skrócić. Jeśli natomiast $r > 1$, to obliczamy $t = p(q'/r) + p'(q/r)$ oraz $s = \text{NWD}(t, r)$, i wtedy:

$$\frac{p}{q} + \frac{p'}{q'} = \frac{pq' + p'q}{qq'} = \frac{t/s}{(q/r)(q'/s)}.$$

Wykonajmy przykładowe obliczenia według obu wzorów dla sumy $15/52 + 19/12$. Zgodnie z pierwszym wzorem otrzymujemy: $(180 + 988)/624 = 1168/624$ i po skróceniu przez $\text{NWD}(1168, 624) = 16$, mamy ułamek $73/39$. Zgodnie z drugim wzorem mamy: $r = \text{NWD}(52, 12) = 4$, $t = 15 \cdot 3 + 19 \cdot 13 = 292$, $s = \text{NWD}(292, 4) = 4$. Stąd: $15/52 + 19/12 = (292/4)/((52/4)(12/4)) = 73/39$. Jak widać, w drugim przypadku wystąpiły znacznie mniejsze liczby niż w pierwszym.

Mnożenie. Wykonujemy: $(p/q)(p'/q') = (pp')/(qq')$ i skracamy licznik i mianownik przez $\text{NWD}(pp', qq')$. Można jednak uprościć te działania. Ponieważ p i q oraz p' i q' są parami względnie pierwszych liczb, mamy więc $\text{NWD}(pp', qq') = rs$, gdzie $r = \text{NWD}(p, q')$ i $s = \text{NWD}(q, p')$. Stąd wynika następująca równość:

$$\frac{p}{q} \cdot \frac{p'}{q'} = \frac{pp'}{qq'} = \frac{(p/r)(p'/s)}{(q/s)(q'/r)}$$

W tej metodzie obliczamy dwa razy NWD , ale dla mniejszych liczb i w praktyce obliczenia nie trwają dłużej.

Dzielenie. Postępujemy podobnie, jak w drugiej metodzie mnożenia ułamków. Niech $r = \text{NWD}(p, p')$ i $s = \text{NWD}(q, q')$. Wtedy metodę dzielenia ułamków zwykłych opisuje wzór, w którym dodatkowo zakładamy, że $p' > 0$:

$$\frac{p}{q} \bigg/ \frac{p'}{q'} = \frac{pq'}{qp'} = \frac{(p/r)(q'/s)}{(q/s)(p'/r)}$$

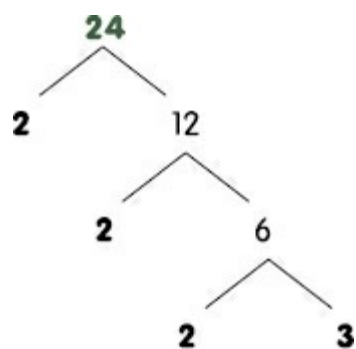
7.6. Liczby pierwsze i liczby złożone

Przypomnijmy najpierw podstawowe definicje. **Liczba** naturalna jest **pierwsza**, jeśli dzieli się tylko przez 1 i przez siebie. Liczbę naturalną, która nie jest pierwszą, nazywa się **liczbą złożoną**. Liczba złożona ma więc **dzielniki**, które są różne od 1 i od niej samej — na przykład 24 nie jest liczbą pierwszą, gdyż jest podzielna przez liczby: 2, 3, 4, 6, 8 i 12. **Czynnikiem**^[1] liczby złożonej jest jej dzielnik, który jest liczbą pierwszą. Zatem czynnikami liczby 24 są tylko liczby 2 i 3, gdyż pozostałe jej dzielniki nie są liczbami pierwszymi. **Rozkładem liczby na czynniki (pierwsze)** lub jej faktoryzacją nazywamy przedstawienie liczby w postaci iloczynu jej czynników pierwszych z uwzględnieniem ich krotności, czyli liczba 24 ma rozkład na czynniki $2 \cdot 2 \cdot 2 \cdot 3 = 2^3 \cdot 3$.

Jest wiele słuszności w powiedzeniu, że wśród liczb naturalnych liczby pierwsze odgrywają rolę podobną do roli pierwiastków w chemii.

Dość wcześnie poznajemy w szkole sposób rozkładu liczby na czynniki. Najczęściej w tym celu znajdujemy kolejne dzielniki liczby, zwykle od najmniejszego, i tworzymy drzewo rozkładu. Na rysunku 7.6 jest przedstawione drzewo rozkładu liczby 24 — czynniki pierwsze z rozkładu znajdują się w liściach tego drzewa.

Rysunek 7.6. Drzewo rozkładu liczby 24 na czynniki



Ćwiczenie 7.10. Opisz algorytm rozkładania danej liczby naturalnej na czynniki, który poznałeś na lekcjach matematyki. ■

Matematycy od dawna starali się podać wzór na liczby pierwsze. Pierre de Fermat (1601-1665) twierdził, że dla każdego k liczba $2^{2^k} + 1$ jest liczbą pierwszą. Niestety, mylił się — dla $k = 5$ nie jest to liczba pierwsza. Najczęściej stosowanym wzorem na liczby pierwsze jest $2p - 1$, gdzie p jest liczbą pierwszą. Na przykład dla $p = 2, 3, 5, 7, 19, 31, 61, 89$ liczby tej postaci są pierwsze. Ten wzór zasugerował ojciec Marin Mersenne (1588 - 1648) w korespondencji z P. Fermatem i liczba pierwsza o tej postaci nazywa się **liczbą Mersenne’a**. Piszemy o tym w książce [Piramidy].

Algorytm rozkładu liczby na czynniki uściślimy i nieco usprawnimy w punkcie

7.6.1.

Rozkładem liczby na czynniki posługujemy się na przykład przy wyznaczaniu NWD i NWW liczb — wspominamy o tym jedynie w zadaniu 7.7, gdyż nie są to najszybsze algorytmy dla komputerów, chociaż bardzo wygodne w obliczeniach na papierze.

W punkcie 7.4 omówiliśmy obliczanie największego wspólnego dzielnika dwóch danych liczb. Na początku tamtego punktu opisaliśmy algorytm znajdowania NWD wśród kolejno sprawdzanych liczb naturalnych. Ta dość oczywista metoda jest jednak bardzo powolna. Podaliśmy następnie algorytm Euklidesa, w którym są wykonywane bardzo proste operacje i dość szybko jest znajdowany NWD.

Znamy zatem prosty algorytm obliczania wspólnego dzielnika dwóch liczb. Niestety, o wiele trudniej jest wyznaczyć dzielniki jednej liczby lub sprawdzić, czy dana liczba jest liczbą pierwszą. Jesteśmy na ogół zdani na metodę, w której sprawdza się podzielność przez liczby z dużego zbioru możliwych dzielników.

Najczęściej są rozważane następujące dwa problemy związane z liczbami pierwszymi:

1. Sprawdzić, czy dana liczba n jest liczbą pierwszą; jeśli nie jest, to podać jej rozkład na czynniki.
2. Znaleźć wszystkie liczby pierwsze w wybranym przedziale liczb lub znaleźć dużą liczbę pierwszą.

Największa znana (w dniu 3 lipca 2016 roku) liczba pierwsza została znaleziona na początku 2016 roku. Jest ona liczbą Mersenne’a dla $p = 74\,207\,281$ i zawiera 22 338 618 cyfr. Aktualne informacje na temat znajdują się na stronie <http://www.mersenne.org/>, polecamy także rozdział 8. w książce [Piramidy].

Algorytmy rozwiązywania tych zadań opisujemy w dwóch następnych punktach — tutaj jeszcze kilka uwag o znaczeniu liczb pierwszych. Są one obecnie wykorzystywane w **kryptografii** do kodowania wiadomości przesyłanych w sieciach komputerowych. W tym celu są potrzebne duże takie liczby. Dlatego obecnie tak wiele uwagi poświęca się znajdowaniu coraz większych liczb pierwszych. Euklidesa (300 rok p.n.e.) już jednak ciekawiło, czy takich liczb jest dużo, i udowodnił następujące twierdzenie, z którego wynika, że liczb pierwszych jest nieskończenie wiele:

Liczb pierwszych jest więcej niż jest ich w jakimkolwiek skończonym zbiorze złożonym z liczb pierwszych.

T

Uzasadnijmy to stwierdzenie Euklidesa. Przypuśćmy, że mamy zbiór złożony z liczb pierwszych $P = \{p_1, p_2, \dots, p_k\}$ i zdefiniujmy liczbę $M = p_1 p_2 \cdots p_k + 1$.

Możliwe są dwa przypadki: albo M jest liczbą pierwszą, albo nie jest liczbą pierwszą. W pierwszym przypadku — M jest więc liczbą pierwszą, która nie należy do zbioru P . Jeśli natomiast liczba M nie jest liczbą pierwszą, to musi być podzielna przez liczbę mniejszą od niej, w szczególności musi być podzielna przez jakąś liczbę pierwszą. Dzielnikiem pierwszym liczby M nie może być jednak żadna z liczb p_1, p_2, \dots, p_k , gdyż te liczby dzielą liczbę $M - 1$, a zatem jest nią liczba pierwsza, która nie należy do zbioru P . To kończy nasze uzasadnienie prawdziwości twierdzenia Euklidesa. ■

Na przykład dla zbioru trzech liczb pierwszych $P = \{2, 3, 7\}$, $M = 2 \cdot 3 \cdot 7 + 1 = 43$ jest liczbą pierwszą. Jeśli natomiast $P = \{2, 3, 7, 43\}$, to $M = 2 \cdot 3 \cdot 7 \cdot 43 + 1 = 1807$ i nie jest to liczba pierwsza, gdyż $1807 = 13 \cdot 139$, ale M dzieli się przez liczbę pierwszą 13, która nie należy do zbioru P .

7.6.1. Rozkład liczby na czynniki pierwsze

Opiszemy tutaj algorytm znajdowania rozkładu liczby naturalnej na czynniki.

Dla danej liczby naturalnej n szukamy następującego jej przedstawienia:

$$n = p_1 p_2 \dots p_l, \quad p_1 \leq p_2 \leq \dots \leq p_l \quad (7.11)$$

gdzie p_i ($i = 1, 2, \dots, l$) jest liczbą pierwszą. Oczywista metoda znajdowania czynników p_i polega na dzieleniu n przez kolejne liczby pierwsze $p = 2, 3, 5, 7, 11, \dots$ itd. Jeśli natrafimy na p takie, że $n \bmod p = 0$, czyli liczba n jest podzielna przez p , to p jest najmniejszym dzielnikiem n . Postępowanie to powtarzamy dla liczby n równej n/p i dla kolejnych liczb pierwszych, począwszy od p — gdyż p może wielokrotnie wystąpić w rozkładzie (7.11). Jeśli w jakimkolwiek momencie sprawdzania podzielności dla kolejnego p okaże się, że liczba n nie jest podzielna przez p i część całkowita ilorazu n/p nie jest większa niż p , to możemy przerwać obliczenia i stwierdzić, że bieżąca liczba n jest liczbą pierwszą. To ostatnie stwierdzenie wynika z następującej własności:

Jeśli n nie jest liczbą pierwszą, to ma czynnik p spełniający warunek $p \leq \sqrt{n}$.

Ponadto, każda liczba złożona ma w swoim rozkładzie na czynniki co najwyżej jeden czynnik większy niż \sqrt{n} . ■

Prawdziwość drugiej części tego stwierdzenia wynika stąd, że gdyby liczba złożona miała dwa czynniki większe od \sqrt{n} , to ich iloczyn w przedstawieniu (7.11) byłby większy od n , czyli wystąpiłaby sprzeczność.

Możemy teraz podać algorytm znajdujący rozkład liczby naturalnej na czynniki, będące liczbami pierwszymi.

Algorytm faktoryzacji liczby naturalnej

Dane: Liczba naturalna n oraz ciąg jej możliwych dzielników: $2 = d_0 < d_1 < d_2 < \dots$, który zawiera wszystkie liczby pierwsze mniejsze lub równe \sqrt{n} oraz

liczbę, która nie jest mniejsza od \sqrt{n} . {Ponieważ znajdowanie ciągu

kolejnych liczb pierwszych jest pracochłonne, w algorytmie stosujemy ciąg dzielników, który oprócz liczb pierwszych może zawierać jeszcze inne liczby.}

Wynik: Rozkład liczby n na czynniki w postaci (7.11) lub informacja, że n jest liczbą pierwszą.

Krok 1. $l := 0$; $i := 0$; {Początkowe wartości indeksów czynników i dzielników.}

Krok 2. Jeśli $n = 1$, to zakończ algorytm.

Krok 3. {Dzielenie przez kolejny dzielnik.} $q := n \text{ div } d_i$; $r := n \bmod d_i$.

Krok 4. Jeśli $r \neq 0$, to przejdź do kroku 6.

Krok 5. {Znaleziony został kolejny czynnik.} $l := l + 1$; $p_l := d_i$; $n := q$.

Powrót do kroku 2.

Krok 6. Jeśli $q > d_i$, to przypisz $i := i + 1$ i przejdź do kroku 3.

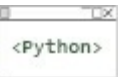
Krok 7. { n jest ostatnim dzielnikiem.} Przypisz $l := l + 1$ oraz $p_l := n$ i zakończ algorytm. ■

Zastosujmy ten algorytm, by rozłożyć liczbę $n = 29529$ na czynniki pierwsze, wykorzystując do tego ciąg możliwych dzielników złożony z kolejnych liczb pierwszych. Liczba 29529 jest podzielna przez 3, a więc w kroku 5. otrzymujemy: $p_1 := 3$, $n := 9843$. W następnej iteracji, ponownie w kroku 5., otrzymujemy: $p_2 := 3$, $n := 3281$. Liczba $n = 3281$ nie jest podzielna przez żadną z liczb pierwszych: 5, 7, 11, 13. Otrzymujemy natomiast $3281 = 17 \cdot 193$, czyli w kroku 5. mamy $p_3 := 17$, $n = 193$. W następnej iteracji dla tego samego dzielnika 17 w kroku 6. okazuje się, że nie jest spełniona nierówność $q = 11 > 17$, czyli bieżąca wartość liczby $n = 193$ nie może mieć dzielników równych co najmniej 17. Zatem w kroku 7. przypisujemy $p_4 := 193$ i kończymy obliczenia. Badana liczba ma więc następujący rozkład na czynniki pierwsze: $n = 29529 = 3 \cdot 3 \cdot 17 \cdot 193$. W algorytmie zostało wykonanych osiem dzieleni liczby rozkładanej (lub jej czynników) przez możliwe dzielniki.

Realizacja algorytmu faktoryzacji liczby naturalnej w obu językach programowania:



```
procedure Faktoryzacja(n:integer; d:Tablica0m; var l:integer; var p:Tablica1m);  
  var i,q,r:integer;  
begin  
  l:=0; i:=0;  
  while n>1 do begin  
    q:=n div d[i];  
    r:=n mod d[i];  
    if r=0 then begin  
      l:=l+1;  
      p[l]:=d[i];  
      n:=q  
    end {r=0}  
    else begin  
      if q>d[i] then i:=i+1  
      else begin  
        l:=l+1;  
        p[l]:=n;  
        n:=1  
      end  
    end {r<>0}  
  end {while n>1}  
end; {Faktoryzacja}
```



```
def Faktoryzacja(n,lista):  
  i=0  
  czynniki=[]  
  while n > 1:  
    q=n // lista[i]  
    r=n % lista[i]  
    if r == 0:  
      czynniki=czynniki+[lista[i]]  
      n=q  
    else:
```

```

    if q > lista[i]:
        i=i+1
    else:
        czynniki=czynniki+[n]
        n=1
return(czynniki)

```

7.6.2. Sito Eratostenesa

Najprostszym sposobem otrzymania wszystkich liczb pierwszych, mniejszych od danej liczby N , jest usunięcie liczb złożonych ze zbioru wszystkich liczb mniejszych od N . By to wykonać systematycznie, usuwamy najpierw liczby podzielne przez 2, później przez 3, następnie podzielne przez 5 itd., aż do osiągnięcia liczby pierwszej p , która spełnia nierówność $p > \sqrt{N}$. Ze

stwierdzenia umieszczonego przed algorytmem faktoryzacji wynika bowiem, że każda liczba złożona nie większa niż N ma dzielnik nie większy niż \sqrt{N} . W tabeli

7.2 jest przedstawiony efekt wykonania czterech pierwszych kroków tej metody. Nazywa się ją **sitem**, gdyż efekt wykreślenia liczb podzielnych przez wybraną liczbę pierwszą można interpretować jako nastawienie oczek sita na takie tylko liczby. Sito, które tutaj omawiamy, podał Eratostenes w III wieku p.n.e.

Tabela 7.2. Efekt działania sita Eratostenesa dla czterech początkowych liczb pierwszych: 2, 3, 5 i 7 na tablicy złożonej ze stu pierwszych liczb naturalnych

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Stosowanie takiej tablicy, zawierającej wszystkie liczby naturalne, jest wygodne w obliczeniach na papierze. W algorytmie, który opiszemy, zredukujemy tę tablicę liczb i wyeliminujemy niektóre działania, opierając się na następujących spostrzeżeniach:

- wystarczy od samego początku rozważać jedynie liczby nieparzyste, gdyż parzyste będą „przesiane” w pierwszym kroku przez oczka nastawione na liczby podzielne przez 2;
- niech p będzie kolejną liczbą pierwszą, której mamy użyć do odsiania liczb złożonych, podzielnych przez p . Wtedy wystarczy wyeliminować liczby mające postać: $p^2, p(p + 2), p(p + 4), \dots$. Wynika to stąd, że liczby qp , gdzie $q < p$, zostały przesiane przez q , a liczb $p(p + 1), p(p + 3), \dots$ nie musimy już przesiewać, gdyż są one parzyste (p , jako liczba pierwsza, jest liczbą nieparzystą).
- kolejnym uproszczeniem jest wykonywanie w algorytmie jedynie dodawań w miejsce mnożeń — kolejno eliminowana liczba jest bowiem o $2p$ większa od poprzedniej.

Algorytm sita Eratostenesa

Dane: Liczba naturalna N (dla uproszczenia założmy, że jest to liczba

parzysta, równa $2M$).

Wyniki: Tablica $S[1..M]$, w której $S_i = 1$, jeśli $2i + 1$ jest liczbą pierwszą, a 0 — w przeciwnym przypadku.

Krok 1. {Początkowe wartości zmiennych.} $S_j := 1$ dla $j = 1, 2, \dots, M$; $i := 1$; $p := 3$; $q := 4$.

Krok 2. Jeśli $S_i = 0$, to przejdź do kroku 4.

Krok 3. {Liczba pierwsza p służy do odsiania jej wielokrotności, począwszy od miejsca q w tablicy S .} Przyjmij: $q := p$. Dopóki $j \leq M$, wykonuj: $S_j := 0$; $j := j + p$.

Krok 4. Wykonaj przypisania: $i := i + 1$; $p := p + 2$; $q := q + 2p - 2$. Jeśli $q \leq M$, to wróć do kroku 2., w przeciwnym razie zakończ algorytm. ■

Zastosujemy teraz tę realizację sita do przesiania liczb pierwszych nie większych niż 100, czyli $N = 2M = 100$. Tabela 7.3 zawiera tylko najważniejsze informacje o działaniu algorytmu w tym przypadku.

Tabela 7.3. Przykład działania sita Eratostenesa dla liczby naturalnej $N = 100$

i	$S[i]$	p	q	Odsiane liczby
1	1	3	4	9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99
2	1	5	12	25, 35, 45, 55, 65, 75, 85, 95
3	1	7	24	49, 63, 77, 91
4	0	9	40	p nie jest liczbą pierwszą
5	1	11	60	$q > M$ — koniec algorytmu

Algorytm sita Eratostenesa w językach Pascal i Python jest zrealizowany w poniższych programach. W tablicy s (procedura w języku Pascal) i w liście $lista$ (procedura w języku Python), i -ty element jest równy 1, jeśli liczba $2i+1$ jest pierwsza.



```
procedure Sito(m:integer; var s:Tablica1m);
var i,j,p,q:integer;
begin
for j:=1 to m do s[j]:=1;
```



```

i:=1; p:=3; q:=4;
repeat
  if s[i]<>0 then begin
    j:=q;
    while j<=m do begin
      s[j]:=0;
      j:=j+p
    end
  end;
  i:=i+1;
  p:=p+2;
  q:=q+2*(p-1)
until q>m
end; {Sito}

```



```

def Sito(m):
    lista=[0]
    for i in range(1,m+1,1):
        lista=lista+[1]
    i=1
    p=3
    q=4
    while q <= m:
        if lista[i] != 0:
            j=q
            while j <= m:
                lista[j]=0
                j=j+p
            i=i+1
            p=p+2
            q=q+2*(p-1)
    return(lista)

```

7.7. Obliczanie wartości pierwiastka kwadratowego

W dotychczas przedstawionych algorytmach iteracja polegała na wykonaniu pewnych operacji albo z góry zadaną liczbę razy (tak było w algorytmach z rozdziału 5. i 6.), albo aż do spełnienia wyróżnionego warunku (jak to jest w algorytmach z tego rozdziału). W tym punkcie rozważamy zadanie, w którego rozwiązaniu liczba iteracji wykonywanych przez algorytm (dla konkretnych danych) zależy od wartości parametru. Tę wartość można ustalić przed wykonaniem obliczeń, by otrzymać rozwiązanie odpowiedniej „jakości”.

Algorytmy o takiej własności są na ogół związane z iteracjami, które zapewniają osiągnięcie wyniku o zadeklarowanej dokładności — wtedy im większą chcemy osiągnąć dokładność obliczeń, tym więcej musimy wykonać iteracji. W tej książce omówimy jeszcze jeden problem (zobacz punkt 9.4), który można rozwiązać w podobny sposób. Obliczenia w tych dwóch przypadkach są zaliczane do działu algorytmiki, zwanego **analizą numeryczną**.

Teraz omówimy sposób obliczania wartości pierwiastka kwadratowego z danej liczby dodatniej, niekoniecznie naturalnej i niekoniecznie będącej pełnym kwadratem innej liczby.

Zadanie 7.2. Przypomnij sobie poznaną na lekcjach matematyki metodę rachunkową obliczania kolejnych cyfr pierwiastka kwadratowego z danej, nieujemnej liczby \sqrt{a} . Opisz tę metodę w postaci algorytmu. Jako dodatkową

daną (parametr) w tym algorytmie przyjmij liczbę cyfr, które chcesz uzyskać w wyniku. ■

Wartość pierwiastka kwadratowego z liczby a można również wyznaczać za pomocą **metody iteracyjnej**, według której — rozpoczynając obliczenia od wybranego rozwiązania początkowego, czyli pewnego przybliżenia szukanej wartości pierwiastka — buduje się ciąg o wartościach zbliżających się do dokładnej wartości \sqrt{a} . Taką metodę można wyprowadzić na podstawie prostej

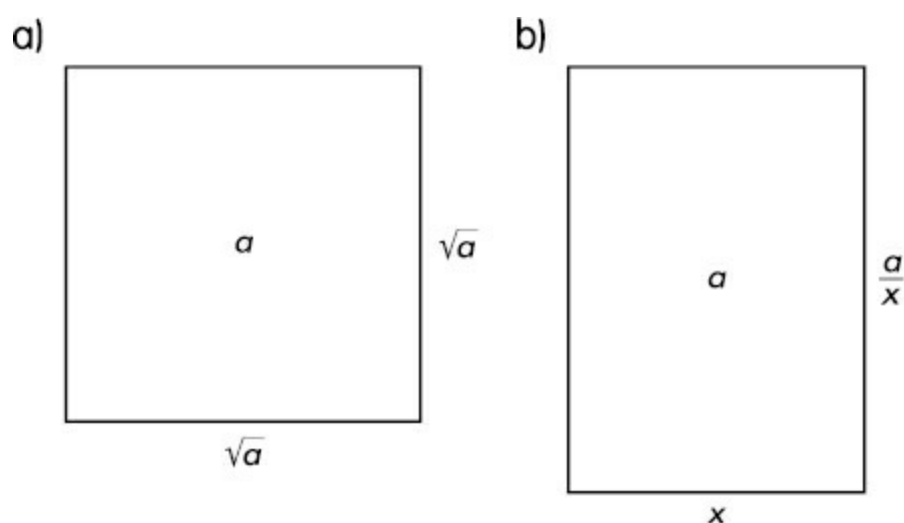
interpretacji geometrycznej pierwiastka kwadratowego — zobacz rysunek 7.7a. Przypuśćmy, że znamy pole kwadratu a i chcemy znaleźć długość jego boku, czyli poszukiwaną przez nas wartość \sqrt{a} . W tym celu możemy wybrać

przybliżoną wartość pierwiastka x , ale wtedy, by pole kwadratu nie uległo zmianie, drugi bok musi być równy a/x . O tym, jak dobre wybraliśmy przybliżenie wartości x , może świadczyć różnica między długościami obu boków prostokąta (zobacz rysunek 7.7b), czyli różnica między x i a/x . Jeśli te wielkości nie są sobie równe, to szukana wartość pierwiastka kwadratowego leży gdzieś między nimi. Za kolejne przybliżenie wartości pierwiastka możemy więc przyjąć liczbę środkową między tymi wartościami, czyli ich średnią arytmetyczną. W ten sposób generujemy ciąg x_0, x_1, \dots , w którym x_0 jest przybliżeniem początkowym,

a kolejne wyrazy są obliczane według następującego wzoru:

$$x_{i+1} = \frac{1}{2} \left(x_i + \frac{a}{x_i} \right) \quad (7.12)$$

dla $i = 1, 2, \dots$, tzn. następne przybliżenie jest równe średniej arytmetycznej z długości boków x i oraz a/x i prostokąta o polu równym a . Aby ten przepis generowania przybliżeń szukanej wartości a stał się algorytmem, musimy jeszcze określić, kiedy zakończyć w nim obliczenia.



Rysunek 7.7. Geometryczna ilustracja do wyprowadzenia algorytmu Newtona-Raphsona

Wyróżnia się dwa kryteria zakończenia obliczeń w metodach iteracyjnych, gdy:

1. Wykonano już podaną przez nas liczbę iteracji.
2. Dwa kolejne przybliżenia leżą dostatecznie blisko siebie — wówczas wśród danych podajemy wartość parametru Eps , którą można nazywać **dokładnością obliczeń**, i sprawdzamy warunek:

$$|x_{i+1} - x_i| \leq Eps \quad (7.13)$$

Jeśli na przykład przyjmiemy $Eps = 10^{-8}$, to spełnienie nierówności (7.13) oznacza, że dwa kolejne przybliżenia szukanej wartości pierwiastka kwadratowego są zgodne aż do ósmego miejsca po kropce. Wtedy możemy przyjąć, że bieżące przybliżenie wartości pierwiastka zgadza się z dokładną wartością pierwiastka w tym samym stopniu. Założona przez nas wartość Eps nie może być zbyt mała; powinna uwzględniać dokładność obliczeń komputerowych. W zadaniu 7.9 podajemy inne kryterium drugiego rodzaju.

Ten algorytm generowania ciągu przybliżeń wartości pierwiastka kwadratowego za pomocą zależności (7.12) jest znany w matematyce pod nazwą **metody Herona**, a w analizie numerycznej — jako **algorytm Newtona-**

Raphsona. Przedstawimy teraz jego realizację, w której stosujemy oba kryteria zakończenia obliczeń — kończą się one, gdy wykonano już *MaxIter* iteracji lub została osiągnięta zadeklarowana dokładność obliczeń *Eps*, według wzoru (7.13).

Algorytm Newtona-Raphsona

Dane: Liczba podpierwiastkowa $a > 0$, początkowe przybliżenie wartości pierwiastka p , maksymalna liczba iteracji *MaxIter*, dokładność obliczeń *Eps*.

Wynik: Przybliżona wartość pierwiastka kwadratowego z a , otrzymana po wykonaniu nie więcej niż *MaxIter* iteracji lub po osiągnięciu dokładności *Eps*.

Krok 1. $Iter := 0$.

Krok 2. $x := (p + a/p)/2$; $Iter := Iter + 1$.

Krok 3. Jeśli $Iter = MaxIter$ lub $|p - x| < Eps$, to zakończ algorytm — x jest szukaną wartością przybliżoną pierwiastka kwadratowego z liczby a .

Krok 4. Przyjmij $p := x$ i wróć do kroku 2. ■

	a =	2,0000000000000000
	i	x[i]
Dane:	0	2,0000000000000000
	1	1,5000000000000000
	2	1,4166666666666670
	3	1,414215686274510
	4	1,414213562374690
	5	1,414213562373090
	6	1,414213562373090
	7	1,414213562373090
	8	1,414213562373090
	9	1,414213562373090
	10	1,414213562373090

Rysunek 7.8. Obliczenia wykonane algorytmem Newtona-Raphsona w arkuszu kalkulacyjnym.

Ćwiczenie 7.11. Na rys. 7.8 jest przedstawionych 10 kolejnych wartości przybliżeń pierwiastka kwadratowego z liczby 2 obliczonych ze wzoru 7.12 dla wartości początkowej 2. Zaobserwuj, że już od piątej iteracji wyniki przybliżenia nie zmieniają się na 15 pierwszych miejscach po kropce. Samodzielnie zapisz w arkuszu te obliczenia i wykonaj je dla różnych liczb podpierwiastkowych i różnych przybliżeń początkowych. ■

Ćwiczenie 7.12. Zapisz algorytm Newtona-Raphsona w języku Pascal lub Python i wykonaj z jego pomocą eksperymenty obliczeniowe, przyjmując dla wybranej liczby podpierwiastkowej a różne wartości: dokładności obliczeń *Eps*, przybliżenia początkowego p oraz maksymalnej liczby iteracji *MaxIter*.

Zaobserwuj, jak zwiększa się liczba iteracji, gdy zwiększasz dokładność (czyli zmniejszasz Eps). Powtórz eksperyment dla przybliżenia początkowego $p = (1 + a)/2$. Jaka jest liczba iteracji wykonywanych w tym przypadku dla różnych dokładności obliczeń, w porównaniu z innymi przybliżeniami początkowymi? Porównaj wyniki otrzymane w arkuszu kalkulacyjnym przy realizacji ćwiczenia 7.11 z tymi, które otrzymałeś, posługując się programem. ■



Zadanie 7.3. Uogólnij rozumowanie prowadzące nas do wzoru (7.12) dla obliczania przybliżonej wartości pierwiastka kwadratowego tak, aby otrzymać wzór na obliczanie przybliżonej wartości pierwiastka trzeciego stopnia. W tym celu posłuż się podobną interpretacją, jak na rysunku 7.7, ale wykonaną w przestrzeni 3-wymiarowej. Postaraj się pójść krok dalej i wyprowadź wzór dla iteracyjnego obliczania przybliżonej wartości pierwiastka dowolnego stopnia k . ■

7.8. Zadania i problemy

Zadanie 7.4. Podstawą w systemie szesnastkowym jest liczba $b = 16$ — system ten często występuje w obliczeniach komputerowych. Zgodnie z definicją systemu pozycyjnego, „cyframi” w nim są „cyfry” szesnastkowe $\{0, 1, 2, \dots, 9, 10, 11, 12, 13, 14, 15\}$. Dla uniknięcia niejednoznaczności w zapisie i nieporozumień w obliczeniach, powszechnie stosuje się następujący układ cyfr szesnastkowych: $\{0, 1, 2, \dots, 9, A, B, C, D, E, F\}$. Na przykład liczbę 44 zapisuje się następująco: $(2C)_{16}$, gdyż $44 = 2 \cdot 16^1 + 12 \cdot 16^0$. Napisz program w którymś z języków, służący do znajdowania rozwinięcia danej liczby w systemie szesnastkowym.

Problem 7.1. Porównaj najpierw rozwinięcia kilku liczb w systemie binarnym i w **systemie ósemkowym**. Opisz zauważoną zależność i sformułuj ją jako ogólną zasadę. Korzystając z tej zasady, sporządź opisy algorytmów: zamiany liczby w systemie binarnym na liczbę w systemie ósemkowym i szesnastkowym oraz zamiany w odwrotną stronę.

Problem 7.2. Opisz algorytm odwracania ciągu. W **ciągu odwróconym** element pierwszy zajmuje miejsce elementu ostatniego, a ostatni — pierwszego, drugi — przedostatniego, a przedostatni — drugiego itd. Zadbaj, aby w Twoim algorytmie ciąg odwrócony był umieszczany w tym samym miejscu, w którym znajduje się ciąg odwracany. Zrealizuj swój algorytm w postaci programu w wybranym języku programowania.

Zadanie 7.5. Sformułuj algorytm, w którym dla danej liczby naturalnej n jest obliczana najmniejsza liczba naturalna k , spełniająca nierówność: $2k \geq n$. Zatem k ma być najmniejszą potęgą liczby 2, większą od liczby n lub jej równą.

Problem 7.3. Wykaż, że liczba mnożeń w algorytmie potęgowania od lewej do prawej wynosi: $\lfloor \log_2 n \rfloor + v(n) - 1$, gdzie $v(n)$ oznacza liczbę bitów równych 1 w binarnej reprezentacji liczby n .

Problem 7.4. Czas wykonania dzielenia w wielu komputerach jest prawie taki sam, jak czas wykonania mnożenia. Przyjmij, że w algorytmie potęgowania liczb możesz używać zarówno mnożeń, jak i dzielen. Ile takich działań trzeba wykonać, aby obliczyć wartość x^{15} ? Na tym przykładzie porównaj pod względem efektywności tę metodę z metodą binarną. Podaj inne przykłady wskazujące na przewagę jednej z tych metod.

W ogólnym przypadku trudno jest podać, kiedy należy wykonać dzielenie, a kiedy mnożenie. Przedstaw propozycję takiego algorytmu i porównaj jego efektywność z efektywnością algorytmów korzystających z binarnej postaci wykładnika.

Zadanie 7.6. Zapisz w wybranej reprezentacji algorytm wyznaczania ilorazu oraz reszty z dzielenia liczby n przez liczbę m według zależności (7.7), w którym dzielenie jest zrealizowane jako ciąg odejmowań.

Zadanie 7.7. Napisz program w wybranym języku programowania, będący realizacją rozszerzonej wersji algorytmu Euklidesa, opisanej w punkcie 7.5.1.

Problem 7.5. Uzasadnij prawdziwość wzorów i metod podanych w punkcie 7.5.2, upraszczających wykonywanie podstawowych działań arytmetycznych na ułamkach zwykłych. Zmodyfikuj te wzory tak, aby były prawdziwe dla dodatnich i ujemnych ułamków zwykłych.

Problem 7.6. Każdą liczbę naturalną l można rozłożyć na czynniki pierwsze i przedstawić w postaci:

$$l = 2^{l_2} 3^{l_3} 5^{l_5} 7^{l_7} 11^{l_{11}} 13^{l_{13}} \dots = \prod_{p-\text{liczba pierwsza}} p^{l_p} \quad (7.13)$$

gdzie wykładniki l_2, l_3, l_5, \dots są jednoznacznie określonymi nieujemnymi liczbami całkowitymi — wynika to z tak zwanego **podstawowego twierdzenie arytmetyki**. Jeśli dana liczba pierwsza nie występuje w rozkładzie liczby na czynniki, to jej wykładnik w tym przedstawieniu jest równy 0.

Sposób obliczania wartości NWD i NWW dla dwóch danych liczb m i n stosowany na lekcjach matematyki polega na skorzystaniu z rozkładów (7.13) tych liczb w postaci ciągów wykładników $m = (m_2, m_3, m_5, \dots)$ i $n = (n_2, n_3,$

n_5, \dots). Podaj ścisły opis tych algorytmów obliczania wartości NWD i NWW.

Problem 7.7. Wyprowadź wzór na kolejne przybliżenie wartości pierwiastka trzeciego stopnia z liczby a . Zastosuj podobne rozumowanie geometryczne, tym razem w przestrzeni trzywymiarowej. Postaraj się uogólnić ten wzór na przypadek obliczeń wartości pierwiastka k -tego stopnia.

Zadanie 7.8. W arkuszu, służącym do znajdowania przybliżonej wartości pierwiastka kwadratowego, wykonaj obliczenia dla liczby podpierwiastkowej a , która jest kwadratem liczby naturalnej lub kwadratem liczby rzeczywistej o skończonym rozwinięciu. Jakie otrzymałeś wyniki dla różnych wartości przybliżenia początkowego oraz parametrów zakończenia obliczeń?

Zadanie 7.9. W algorytmie Newtona-Raphsona, jako drugie kryterium zakończenia obliczeń, przyjmij spełnienie następującej nierówności:

$$|a - x_{i+1}| \leq Eps,$$

czyli kończ obliczenia, gdy kwadrat kolejnego przybliżenia wartości pierwiastka różni się od liczby podpierwiastkowej nie więcej niż o Eps . Zmodyfikuj odpowiednio arkusz, wypisując wartości wyrażenia z lewej strony powyższej nierówności.

Miałeś okazję dowiedzieć się, że:

- ▶ liczba zapisana w pamięci komputera zajmuje niewiele bitów w porównaniu ze swoją wartością;
- ▶ zapis liczb w różnych **systemach pozycyjnych** ma postać wielomianu;
- ▶ **schemat Hornera** jest najszybszym algorytmem obliczania wartości wielomianu i obliczania dziesiętnych wartości liczb zapisanych w systemie binarnym, ósemkowym lub szesnastkowym;

oraz poznać:

- ▶ szybkie **algorytmy podnoszenia liczb do potęgi**;
- ▶ działanie **algorytmu Euklidesa** w różnych zastosowaniach: wykonywanie obliczeń na liczbach całkowitych i na **ułamkach zwykłych**;
- ▶ szybkie **algorytmy rozkładu liczby** na czynniki pierwsze oraz **odsiewania** liczb złożonych;
- ▶ algorytm wyznaczający rozwiązanie przybliżone **metodą iteracyjną**.

[1] Na ogół czynnikiem liczby nazywa się dowolny jej dzielnik, tutaj jednak przyjmujemy, że **czynnik jest liczbą pierwszą**, gdyż interesują nas jedynie rozkłady liczb na czynniki, które są liczbami pierwszymi.

Rozdział 8. Algorytmy rekurencyjne

Tu poznasz:

- ▶ ogólny schemat rozwiązywania problemów z wykorzystaniem istniejących rozwiązań;
- ▶ pojęcie **rekurencji**;
- ▶ rekurencję tkwiącą w iteracji;
- ▶ **rekurencyjny** sposób **zapisu iteracji**;
- ▶ dwa klasyczne problemy o rekurencyjnej naturze: **Wieża Hanoi** i **liczby Fibonacciego**;
- ▶ **iteracyjny** sposób **zapisu rekurencji**;
- ▶ komputerową realizację algorytmów rekurencyjnych.

Z analizy problemów i sposobów ich rozwiązywania, podanych w poprzednich rozdziałach, wynika, że rozwiązywanie problemu przebiega na ogół w następujących etapach:

- 1.** Najpierw sprawdzamy, czy istnieje gotowe rozwiązanie — jeśli tak, to korzystamy z niego. Problemy, dla których znamy algorytmy, rozwiązujemy na ogół automatycznie, nie zastanawiając się nawet nad stosowaną metodą. Takimi problemami, występującymi w poprzednich rozdziałach, są: znajdowanie największego i najmniejszego elementu w zbiorze, wyłanianie najlepszego gracza w turnieju, porządkowanie kilku liczb. Powszechnie znane sposoby rozwiązywania tych problemów uzupełniliśmy opisami ich własności, tworząc w ten sposób podstawy do ich wykorzystania w szerszym zakresie.
- 2.** Jeśli nie znamy gotowego rozwiązania, staramy się odszukać podobny problem, dla którego rozwiązanie jest nam znane. Wtedy naturalnym pierwszym krokiem na drodze do rozwiązania nowego problemu jest próba zaadaptowania znanego już sposobu rozwiązania. Dobrym przykładem może być tutaj sytuacja, w której — znając algorytm uporządkowania trzech liczb — stosujemy go do porządkowania czterech liczb, umieszczając czwartą wśród już uporządkowanych trzech (zobacz punkt 4.2).
- 3.** Gdy jednak stwierdzamy, że nie możemy skorzystać z gotowych rozwiązań, próbujemy podzielić problem na podproblemy lub tylko wydzielić fragmenty problemu, które potrafimy rozwiązać i staramy się rozbudować te częściowe rozwiązania do algorytmu rozwiązywania całego problemu. Tak postępujemy na przykład w algorytmie jednoczesnego znajdowania największego i najmniejszego elementu w zbiorze (punkt 5.5): najpierw zbiór elementów jest dzielony na dwa podzbiory, a dalsza część algorytmu polega na użyciu znanych algorytmów

znajdowania największego i najmniejszego elementu.

Opisane obok podejście do rozwiązywania problemów jest podstawową metodą pracy matematyków. Jak ujął to ktoś humorystycznie, jeśli matematyk wie, jak wziąć książkę z półki, a przez nieuwagę książka spadła mu na podłogę, to by ją podnieść, stara się sprowadzić ten problem do znanej mu czynności: najpierw więc odkrywa, jak położyć książkę na półce, a następnie stosuje... znane mu już rozwiązanie.

W kolejności tych trzech etapów: korzystanie z istniejących rozwiązań — modyfikowanie i rozszerzanie znanych rozwiązań — łączenie rozwiązań częściowych w rozwiązanie całego problemu, znajdują potwierdzenie skłonności człowieka do upraszczania sposobów rozwiązywania problemów, które przed nim stają. Co więcej — powinniśmy w tym dostrzec również normalne dążenie do korzystania z dotychczasowych zdobyczy i osiągnięć, własnych i innych ludzi.

Ćwiczenie 8.1. W poprzednich rozdziałach, w wielu problemach można odnaleźć inne problemy tam omówione. Podobnie w wielu algorytmach zawarliśmy inne algorytmy lub ich fragmenty, a w programach — inne programy, zwane podprogramami. Postaraj się podać kilka innych niż wymienione wyżej przykładów takich problemów i algorytmów. ■

Podprogramy w języku Pascal przyjmują postać procedur lub funkcji. Wszystkie realizacje algorytmów w języku Pascal załączone do tej książki mają właśnie taką postać. Programy w języku Python prezentujemy na ogół w postaci funkcji.

W tym rozdziale przedstawiamy kolejną możliwość, z której korzysta się przy projektowaniu algorytmów i programów, może nieco zaskakującą na pierwszy rzut oka. Spotkaliśmy już w poprzednich rozdziałach przykłady problemów i ich algorytmów, w których w naturalny sposób można znaleźć części podobne do... całości (zobacz zadania 8.2 – 8.4). W następnym punkcie właśnie w taki sposób spojrzymy na znane z poprzednich rozdziałów problemy i przeformułujemy ich algorytmy tak, by korzystały... z siebie samych. Takie algorytmy nazywamy **rekurencyjnymi**. W punkcie 8.2 stosujemy to podejście do dwóch klasycznych problemów, w których rekurencja tkwi naturalnie. W następnych rozdziałach posługujemy się rekurencją jako jedną z najważniejszych metod konstruowania algorytmów i budowania programów.

Nawet nie zdajemy sobie sprawy, jak często stosujemy rekurencję w potocznym języku. Gdy tłumaczymy komuś, na czym polega „tańczenie tanga”, to mówimy: „zrób krok i tańcz dalej”. Andrzej Jerszow podał przykład rekurencji, który należy już do klasycznych przykładów w informatyce: „jeść kaszkę” to „zjeść łyżkę kaszki i jeść kaszkę dalej”. W pierwszym przypadku tańczymy tak długo, jak długo gra muzyka, a w drugim — jemy, dopóki jest jeszcze kaszka na talerzu.

Zwróćmy uwagę na bardzo krótkie i eleganckie zapisy algorytmów w postaci rekurencyjnej w językach Pascal i Python. Pamiętajmy jednak, że komputerowe

realizację tych algorytmów nie zawsze są najbardziej efektywne.

8.1. Inne spojrzenie na iterację

Rozważania w tym punkcie są innym ujęciem trzech problemów, które rozwiązywaliśmy w poprzednim rozdziale, stosując iterację. Jest to jednocześnie kontynuacja tamtych rozważań i wprowadzenie do rekurencyjnego rozwiązywania problemów.

8.1.1. Obliczanie wartości wielomianu

W punkcie 7.2, opisując przejście od ogólnej postaci wielomianu (7.4) do schematu Hornera (7.5), podaliśmy następującą postać wielomianu stopnia n dla $n \geq 0$:

$$w_n(x) = (a_0 x^{n-1} + a_1 x^{n-2} + \dots + a_{n-1})x + a_n,$$

którą można również zapisać jako:

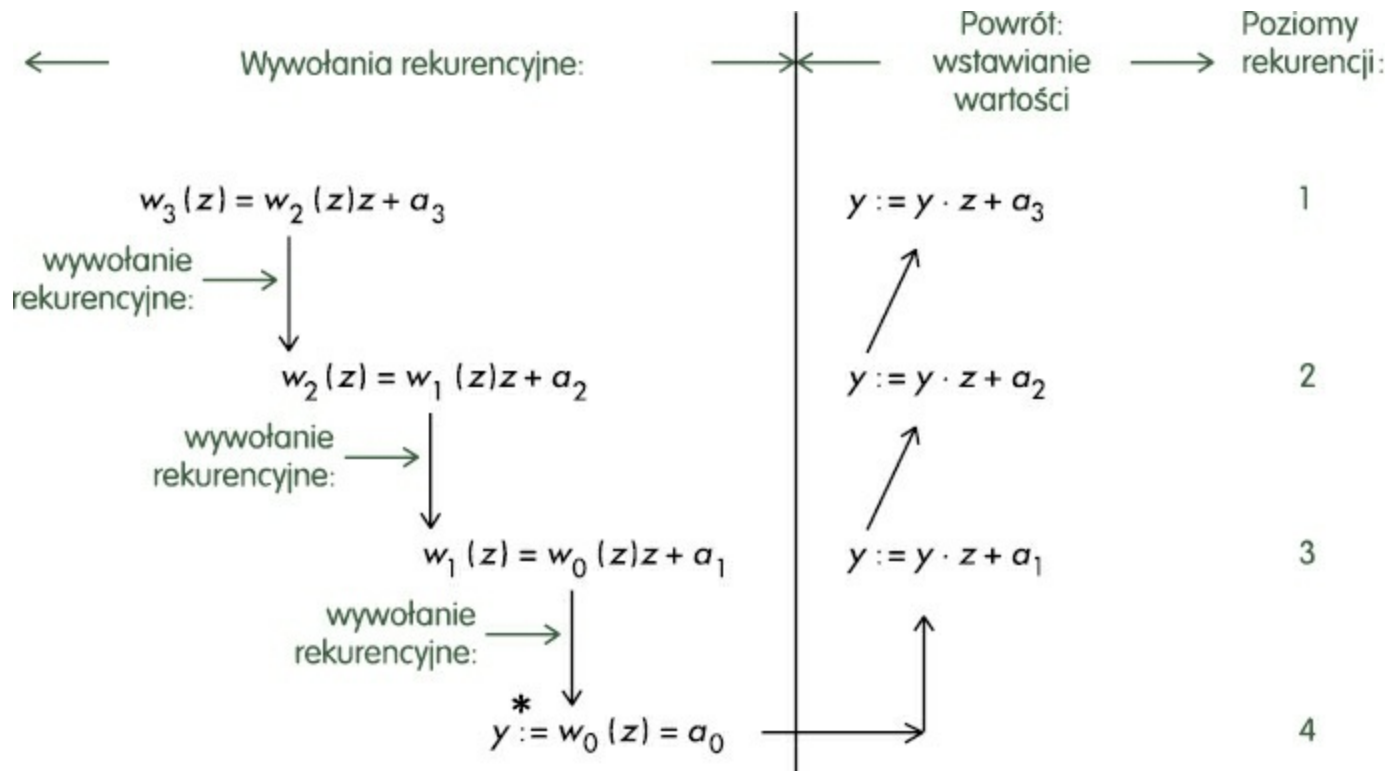
$$w_n(x) = w_{n-1}(x)x + a_n, \quad (8.1)$$

gdzie $w_{n-1}(x) = a_0 x^{n-1} + a_1 x^{n-2} + \dots + a_{n-1}$. Ze wzoru (8.1) wynika, że wartość wielomianu stopnia n może być obliczona, gdy tylko znamy wartość odpowiedniego wielomianu stopnia $n-1$ w tym samym punkcie. Sprawdźmy, czy to spostrzeżenie jest prawdziwe dla każdego n . Dla $n=1$, na podstawie wzoru (8.1), mamy $w_1(x) = w_0(x)x + a_1$, gdzie $w_0(x) = a_0$, a więc poprawnie.

Natomiast dla $n=0$ po prawej stronie w równości (8.1) pojawia się wielomian stopnia -1 , a takiego nie znamy. Zatem dla $n=0$ nie możemy korzystać ze wzoru (8.1), przyjmujemy więc dodatkowo, że $w_0(x) = a_0$. Z tej dyskusji wynika, że wielomian stopnia n można zapisać w następującej postaci:

$$w_n(x) = \begin{cases} a_0, & n=0 \\ w_{n-1}(x) + a_n, & n \geq 1 \end{cases} \quad (8.2)$$

Otrzymaliśmy zależność, w której wielkości definiowanej (wartości wielomianu stopnia n) odpowiada wyrażenie zawierające tę samą wielkość, ale dla parametru mniejszego (tj. wartość wielomianu stopnia $n-1$ dla tego samego argumentu). Dodatkowo dla $n=0$ jest podana wprost wartość definiowanej wielkości — jest to **warunek zakończenia rekurencji**, gdyż dzięki niemu każdy ciąg odwołań do tej samej wielkości ma swój koniec (lub, jak kto woli, swój początek). Taka zależność nazywa się **zależnością rekurencyjną**.



Rysunek 8.1. Schematyczne przedstawienie kolejności wykonywania działań w rekurencyjnym algorytmie obliczania wartości wielomianu trzeciego stopnia

Sposób obliczania wartości wielomianu stopnia n na podstawie zależności rekurencyjnej (8.2) ilustrujemy na rysunku 8.1 dla $n = 3$ i dla $x = z$. Wielomian ma następującą postać: $w_3(z) = w_2(z)z + a_3$. Najpierw, korzystając z drugiej części zależności (8.2), obliczamy wartość wielomianu dla $n = 2$ w tym samym punkcie: $w_2(z) = w_1(z)z + a_2$, potem wartość wielomianu dla $n = 1$: $w_1(z) = w_0(z)z + a_1$. Natomiast dla $n = 0$, z pierwszej części zależności (8.2), mamy $w_0(z) = a_0$. Teraz dopiero z tą wartością wielomianu stopnia 0 możemy wrócić do poprzedniej zależności, a następnie do jeszcze wcześniejszych.

Zatem, w obliczeniach na podstawie zależności rekurencyjnej można wyróżnić dwa etapy:

1. Wywołania rekurencyjne — kończy go skorzystanie z warunku zakończenia rekurencji.
2. Powrót z kolejnych wywołań — obejmujący wykonywanie właściwych obliczeń lub tylko przenoszenie wyników.

Przykład na rysunku 8.1 wskazuje, że obliczenia wykonywane w trakcie powrotu z wywołań rekurencyjnych są w gruncie rzeczy realizacją iteracyjnego algorytmu Hornera. Można stąd wysnuć dwa wnioski, przynajmniej w tym przypadku: rekurencja jest innym sposobem realizacji iteracji i jest przy tym sposobem bardziej rozrzutnym, gdyż pierwszy etap jest jakby zbędny w porównaniu z rozwiązaniem iteracyjnym.

Ćwiczenie 8.2. Posługując się zależnością (8.2), oblicz wartość wielomianu $3x^4$

– $4x^2 + 4x - 5$ w punkcie $x = -2$. ■

Realizacja algorytmu rekurencyjnego wymaga zapisania go w postaci funkcji, która może wywoływać samą siebie. Ponadto ta funkcja powinna zawierać parametr określający **poziom rekurencji**, czyli stopień zagłębienia się wywołań. W przypadku obliczania wartości wielomianu, tym parametrem jest n — stopień wielomianu. Innym parametrem może być wartość argumentu x , dla której obliczamy wartość wielomianu.

W funkcji rekurencyjnej HornerRek, na początku jest sprawdzana wartość n . To sprawdzanie wartości n w funkcji rekurencyjnej musi występować przed wywołaniem rekurencyjnym, inaczej ciąg wywołań rekurencyjnych mógłby się nigdy nie skończyć. Jeśli $n = 0$, to wartości funkcji y jest przypisywana (w języku Pascal) lub jest zwracana (w języku Python) wartość współczynnika $a[0]$, a w przeciwnym razie wywoływana jest rekurencyjnie ta sama funkcja, ale z wartością parametru o jeden mniejszą, czyli $n - 1$.

Elegancja i zwartość zapisu rekurencji jest najlepiej widoczna w tekstach opisu algorytmu rekurencyjnego w obu językach programowania.



```
function HornerRek(n:integer; x:real):real;
begin
  if n = 0 then HornerRek := a[0]
  else HornerRek := HornerRek(n-1,x) * x + a[n]
end; {HornerRek}
```



```
def HornerRek(n,a,x):
  if n == 0:
    return a[0]
  else:
    return HornerRek(n-1,a,x) * x + a[n]
```

8.1.2. Obliczanie potęgi

W punkcie 7.3.2 rozważaliśmy sposoby obliczania wartości potęgi x^m i przedstawiliśmy dwa algorytmy, w których korzysta się z binarnej postaci wykładnika. W pierwszym z nich jest stosowane dodatkowo przedstawienie binarnego rozwinięcia wykładnika m w postaci schematu Hornera. Tutaj podamy zależność rekurencyjną dla potęgi, która jest zrealizowana w drugim z

tych algorytmów, natomiast prowadzi do obliczeń rekurencyjnych według pierwszego z nich. Jak to jest możliwe? Zawdzięczamy to rekurencji — ale po kolei.

Rekurencyjna definicja potęgi zawiera: wartość potęgi dla wykładnika równego 1, co stanowi warunek zakończenia rekurencji, oraz uwzględnia parzystość wykładnika, a więc jest dokładnie tak, jak w algorytmie obliczania wartości potęgi algorytmem „od prawej do lewej”:

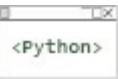
$$x^m = \begin{cases} x, & m=1 \\ \left(x^{m/2}\right)^2, & m - \text{parzyste} \\ \left(x^{(m-1)/2}\right)^2 x, & m - \text{nieparzyste} \end{cases} \quad (8.3)$$

Obliczanie wartości potęgi dla $m = 22$, według wzoru (8.3), przebiega w następującej kolejności. W pierwszym etapie jest redukowana wielkość wykładnika: $(x^{11})^2$, $((x^5)^2 x)^2$, $((x^2)^2 x)^2 x^2$, $((x^2)^2 x)^2 x^2$, a w drugim — jest obliczana wartość potęgi, począwszy od największego zagłębienia nawiasów. Jeśli teraz porównamy ten sposób obliczania wartości potęgi z dwoma algorytmami podanymi w punkcie 7.3.2, to algorytm „od prawej do lewej” działa tutaj na pierwszym etapie tworzenia reprezentacji binarnej wykładnika, a algorytm „od lewej do prawej” wykonuje właściwe obliczenia dla tak znalezionej reprezentacji wykładnika. Zaletą rekurencji w tym przypadku jest to, że nie musimy specjalnie przechowywać binarnej reprezentacji wykładnika — robi to za nas komputer.

Podobnie jak w przypadku rekurencyjnej realizacji schematu Hornera, zapis rekurencyjnego potęgowania w językach Pascal i Python jest bardzo czytelny i stanowi wierny zapis wzorów (8.3).



```
function PotegaRek(m:integer; x:real):real;  
begin  
  if m = 1 then PotegaRek := x  
  else begin  
    polPot = PotegaRek(m div 2,x);  
    if m%2 = 0 then PotegaRek = polPot * polPt  
    else PotegaRek = polPot * polPt * x  
  end; {HornerRek}
```



```
def PotegaRek(m,x):  
    if m == 1:  
        return x  
    else:  
        polPot = PotegaRek(m//2,x)  
        if m%2 == 0:  
            return polPot * polPot  
        else:  
            return polPot * polPot * x
```

8.1.3. Algorytm Euklidesa

Wyprowadzając algorytm Euklidesa w punkcie 7.4, podaliśmy najpierw zależność (7.7) między dzielną, dzielnikiem i resztą, a potem na podstawie tej zależności wysnuliśmy wnioski co do sposobu obliczania NWD dla dwóch liczb m i n . Te wnioski można zapisać w następującej postaci, przy czym przyjmujemy, że $m \leq n$:

$$NWD(m,n) = \begin{cases} n, & m=0 \\ NWD(n \bmod m, m), & m>0 \end{cases} \quad (8.4)$$

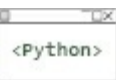
Jest to również **zależność rekurencyjna**, gdyż wartość NWD dla liczb m i n wyraża się wartością tej samej funkcji dla mniejszych argumentów $n \bmod m$ i m . Mamy więc na przykład: $NWD(12, 18) = NWD(6, 12) = NWD(0, 6) = 6$.

Podana w punkcie 7.4 realizacja algorytmu Euklidesa polega na iteracyjnym przekształcaniu pary (m, n) w parę $(n \bmod m, m)$, aż do osiągnięcia reszty $n \bmod m$ równej 0 — wtedy można skorzystać z pierwszej części zależności (8.4) i podać wartość NWD, która jest również wartością NWD dla liczb danych na początku. Przedstawiamy poniżej rekurencyjną realizację algorytmu Euklidesa w postaci funkcji w językach Pascal i Python.



```
function EuklidRek(m,n: integer):integer;  
begin  
    if m > n then EuklidRek := EuklidRek(n,m)  
    else if m = 0 then Euklidrek := n
```

```
else EuklidRek := EuklidRek(n mod m, m)
end; {EuklidRek}
```



```
def EuklidRek(m,n):
    if m > n:
        return EuklidRek(n,m)
    else:
        if m == 0:
            return n
        else:
            return EuklidRek(n % m,m)
```



Polecamy lekturę rozdziału 7. w książce [Piramidy], poświęconego algorytmowi Euklidesa oraz jego modyfikacjom i zastosowaniom. Polecamy również pracę [Log], w której objaśniamy elementarnie, ile operacji podstawowych jest wykonywanych w algorytmie Euklidesa. ■

8.2. Problemy z rekurencyjną naturą

Przykłady zależności i algorytmów rekurencyjnych z poprzedniego punktu mogą stworzyć wrażenie, że rekurencja jest jedynie innym sposobem zapisu iteracji. Co więcej, sposobem mniej jawnym, a przez to trudniejszym do uchwycenia operacji wykonywanych w takich algorytmach. Rzeczywiście tak jest — zamiana algorytmu iteracyjnego na rekurencyjny daje zazwyczaj bardziej zwarty opis realizacji obliczeń, często mający niezaprzeczalne cechy elegancji, ale ta zwartość często ukrywa prostotę obliczeń lub prowadzi do obliczeń bardziej rozrzutnych pod względem liczby wykonywanych operacji.

Podaliśmy te przykłady jako wprowadzenie do myślenia w kategoriach operacji, które ze swej natury mają rekurencyjny charakter, zatem można je prosto zapisać w postaci zależności rekurencyjnych. Ma to jeszcze jeden, nadrzędny cel — by przybliżyć dość trudne pojęcie rekurencji, bo chociaż w większości sytuacji można się bez niej obejść w projektowaniu algorytmów, to jednak algorytmika, jako dziedzina intelektualnych zmagania, straciłaby wiele na swym pięknie, gdyby pozbawić ją możliwości rekurencyjnego wnioskowania, postępowania i obliczeń.

Przedstawimy teraz dwa klasyczne problemy-łamiągłówki, których najbardziej naturalne rozwiązania zapisuje się zwykle w postaci rekurencyjnej, chociaż, jak pokażemy, można je również łatwo rozwiązywać, stosując iterację. Postąpimy więc odwrotnie niż w poprzednim punkcie — najpierw podamy rozwiązanie rekurencyjne, a później przekształcimy je w rozwiązanie iteracyjne.

8.2.1. Wieże Hanoi

Tą nazwą opatruje się łamiągłóvkę, która jest klasycznym przykładem problemu algorytmicznego i służy w informatyce jako ilustracja wielu pojęć i metod rozwiązywania problemów, w tym zwłaszcza rekurencji. Mamy trzy paliki (zobacz pierwszy diagram na rysunku 8.2) — oznaczmy je A, B i C — oraz pewną liczbę krążków różnej wielkości z otworami, nanizanych na palik A w kolejności od największego do najmniejszego, największy znajduje się na dole. Łamiągłówka polega na przeniesieniu wszystkich krążków z palika A na palik B, z możliwością posłużenia się przy tym palikiem C, w taki sposób, że:

- pojedynczy ruch polega na przeniesieniu jednego krążka między dwoma palikami;
- w żadnej chwili rozwiązywania łamiągłówki większy krążek nie może leżeć na mniejszym.

Łamiągłóvkę z Wieżami Hanoi zaproponował w XIX wieku Édouard Lucas, kładąc 8 krążków na paliku A. Odwoływał się przy tym do legendy pochodzącej z Tybetu, która mówi o mnichach rozwiązujących tę łamiągłóvkę z 64 krążkami. Podobno po uporaniu się przez nich ze wszystkimi krążkami ma nastąpić koniec świata!

Jeśli na paliku A znajduje się jeden krążek, to wystarczy przenieść go na palik B, a więc w tym przypadku wystarczy jeden ruch. Jeśli na paliku A są dwa krążki, to łamiągłówka również nie jest trudna: krążek górny przenosimy na palik C, dolny na B i krążek z C przenosimy na B. Zatem wykonujemy w tym przypadku 3 ruchy.

Ćwiczenie 8.3. Spróbuj teraz rozwiązać tę łamiągłóvkę z trzema krążkami. ■

To ćwiczenie nie powinno jednak sprawić Ci większego kłopotu. Czy potrafisz sprecyzować zasadę, którą się posługiwałeś przy przemieszczaniu krążków? Czy wiedząc, jak przenosi się trzy krążki, potrafiłbyś zaproponować sposób rozwiązania tej łamiągłówki z czterema krążkami?

Ostatnie pytanie jest nieco podchwytliwe, bo odnosi się jakby do rozwiązania rekurencyjnego — gdy wiemy, jakie jest rozwiązanie dla trzech krążków, chcielibyśmy umieć rozwiązywać tę łamiągłóvkę z liczbą krążków o jeden większą, korzystając przy tym być może z rozwiązania dla trzech krążków.

Rozwiązanie rekurencyjne

By naprowadzić na rozwiązanie rekurencyjne, zastanówmy się, jaki powinien być układ krążków, gdy wreszcie możemy największy z nich przenieść z palika A na palik B — pamiętajmy przy tym, że tego największego krążka nie możemy położyć na żadnym innym, gdyż wszystkie pozostałe krążki są od niego mniejsze. Zatem, gdy możemy go już przenieść, wszystkie pozostałe krążki muszą być ułożone na paliku C zgodnie z ich wielkością. Stąd wynika, że możliwe rozwiązanie składa się z trzech etapów — zapiszemy je dla dowolnej liczby n krążków na paliku A:

1. Przenieś $n - 1$ górnych krążków z palika A na palik C, używając palika B.
2. Przenieś największy krążek z palika A na palik B.
3. Przenieś wszystkie krążki z palika C na palik B, używając palika A.

Zatem, jeśli umiemy rozwiązać tę łamigłówkę z trzema krążkami, to powyższą metodą możemy znaleźć rozwiązanie dla czterech krążków, na tej podstawie — dla pięciu krążków itd. Możemy skorzystać z tej zasady również w przypadku trzech krążków, gdyż wiemy, jak przenosi się dwa krążki. Powstaje jednak pytanie, czy opisane wyżej kroki mogą być zawsze wykonane? Krok 2. już objaśniliśmy. Kroki 1. i 3. są podobne, a ich wykonalność wynika stąd, że mamy do pełnej dyspozycji trzy paliki, gdyż palik zawierający największy krążek może być również swobodnie wykorzystany przez wszystkie pozostałe krążki. Stosując więc wnioskowanie indukcyjne, możemy być pewni, że kroki 1. i 3. mogą być również wykonane. Powyższy opis posłuży nam teraz do zapisania rekurencyjnego rozwiązania łamigłówki Wież Hanoi w postaci algorytmu.

Algorytm rekurencyjny rozwiązywania łamigłówki Wież Hanoi

Dane: Trzy paliki A, B i C oraz n krążków o różnych średnicach, nanizanych od największego do najmniejszego na palik A. Krążki można przenosić między palikami tylko pojedynczo i nigdy nie można położyć większego na mniejszym.

Wynik: Krążki nanizane na palik B — do uzyskania tego wyniku można wykonywać jedynie dopuszczalne przenoszenia.

Krok 1. Jeśli $n = 1$, to przenieś krążek z palika A na palik B i zakończ algorytm dla $n = 1$.

Krok 2. {W tym przypadku liczba krążków na paliku A jest większa od 1.}

2a. Stosując ten algorytm, przenieś $n - 1$ krążków z A na C, używając B.

2b. Przenieś pozostały krążek z A na B.

2c. Stosując ten algorytm, przenieś $n - 1$ krążków z C na B, używając

A. ■

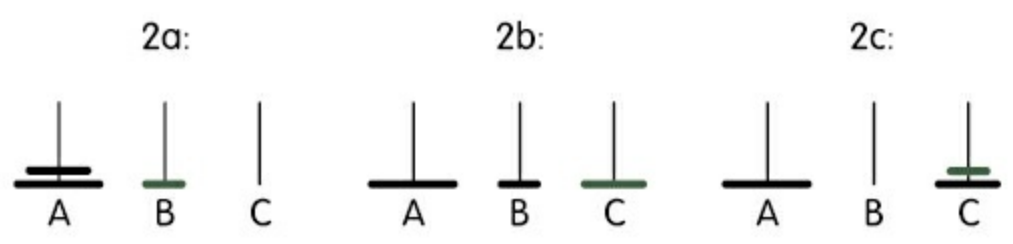
Na rysunku 8.2 przedstawiamy ułożenia krążków na palikach w trakcie wykonywania algorytmu rekurencyjnego dla $n = 3$. Zielonym kolorem są oznaczone krążki przenoszone w danym kroku. Jeśli więcej niż jeden krążek jest zielony, to oznacza, że do przeniesienia tych krążków musi być wykonany krok rekurencyjny.



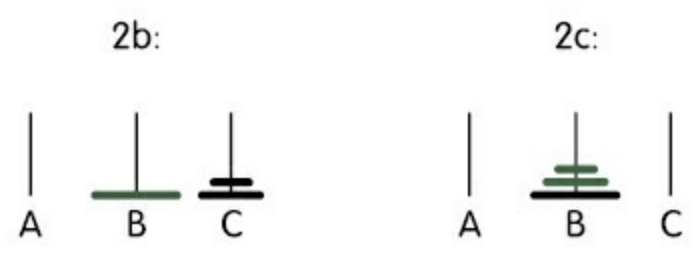
1. poziom rekurencji – krok 2a:



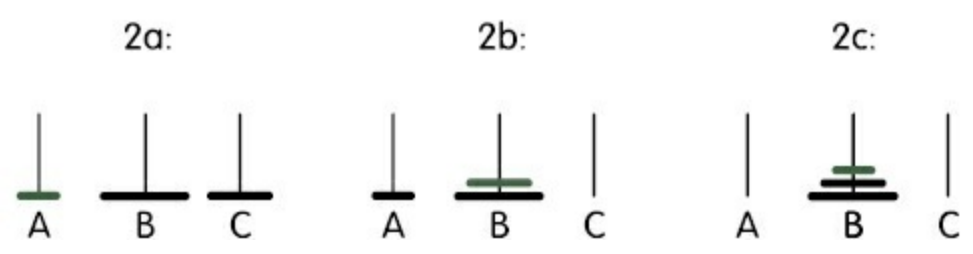
2. poziom rekurencji dla kroku 2a – kroki:



1. poziom rekurencji – kroki:



2. poziom rekurencji dla kroku 2c – kroki:



Rysunek 8.2. Przykład działania rekurencyjnego algorytmu rozwiązywania łamigłówki Wież Hanoi dla $n = 3$. Zielonym kolorem są zaznaczone krążki przenoszone w danym kroku

Aby bardziej szczegółowo opisać realizację powyższego algorytmu, oznaczmy

przez $(X \rightarrow Y)$ przeniesienie krążka z palika X na palik Y, a przez (k, X, Y, Z) — przeniesienie k krążków z palika X na palik Y z wykorzystaniem palika Z, gdzie X, Y, Z oznaczają różne paliki spośród A, B i C. Przy tych oznaczeniach powyższy algorytm można zapisać następująco:

Algorytm rekurencyjny rozwiązywania łamigłówki Wież Hanoi (n, A, B, C)

Krok 1. Jeśli $n = 1$, to $(A \rightarrow B)$ i zakończ algorytm dla tego przypadku.

Krok 2. {W tym przypadku liczba krążków na paliku A jest większa od 1.}

2a. Zastosuj ten algorytm dla $(n - 1, A, C, B)$.

2b. Przenieś pozostały krążek $(A \rightarrow B)$.

2c. Zastosuj ten algorytm dla $(n - 1, C, B, A)$. ■

Można poczynić jeszcze dalsze uproszczenia w tym algorytmie. Zauważmy, że w krokach 1. i 2b. jest wykonywane takie samo przeniesienie krążka, krok 1. jest więc szczególnym przypadkiem kroku 2., w którym dla $n = 1$ przyjmujemy, że kroki 2a. i 2c. są puste (czyli nic nie jest wykonywane). Ostatecznie otrzymujemy następujący algorytm rekurencyjny:

Algorytm rekurencyjny rozwiązywania łamigłówki Wież Hanoi (n, A, B, C)

Krok 1. Jeśli $n = 0$, to nic nie rób i zakończ algorytm dla tego przypadku.

Krok 2. {W tym przypadku liczba krążków na paliku A jest większa od 0.}

2a. Zastosuj ten algorytm dla $(n - 1, A, C, B)$.

2b. Przenieś pozostały krążek $(A \rightarrow B)$.

2c. Zastosuj ten algorytm dla $(n - 1, C, B, A)$. ■

Opisy dwóch ostatnich wersji algorytmu rozwiązywania łamigłówki Wież Hanoi są w pełni rekurencyjne. Podobnie jak w zależności rekurencyjnej, opis algorytmu rekurencyjnego zawiera układ parametrów, dla których jest określone jego działanie, dzięki temu wewnątrz algorytmu można wywoływać ten sam algorytm z odpowiednio dobranymi wartościami tych parametrów.



```
procedure HanoiRek(n:integer; A,B,C:char);
```

```
begin
```

```
  if n > 0 then begin
```

```
    HanoiRek(n-1,A,C,B);
```

```
    writeln('Przenies ',n,' z ',A,' na ',B);
```

```
    HanoiRek(n-1,C,B,A)
```

```
end
```

```
end; {HanoiRek}
```



```
def przenieś(n, skąd, dokąd, pom):  
    if n == 1:  
        print("Przenieś dysk z ", skąd, " na ", dokąd)  
    else:  
        przenieś(n-1, skąd, pom, dokąd)  
        przenieś(1, skąd, dokąd, pom)  
        przenieś(n-1, pom, dokąd, skąd)  
  
def HanoiRek(n):  
    przenieś(n, "A", "B", "C")
```

Rozwiązanie iteracyjne

Rekurencyjne rozwiązanie łamigłówki z trzema krążkami, zobrazowane na rysunku 8.2, pokazuje jednocześnie, w jakiej kolejności są przenoszone pojedyncze krążki — są to kolejne (od góry i od lewej) diagramy złożone z trzech palików, na których na zielono jest oznaczony tylko jeden krążek. Ta kolejność jest iteracyjnym rozwiązaniem łamigłówki Wież Hanoi dla $n = 3$. Czy można stąd wywnioskować, jaką postać ma algorytm iteracyjny dla dowolnej liczby krążków? Przyglądając się uważnie kolejnym ułożeniom krążków na palikach — można, chociaż nie jest to takie łatwe. Zapiszmy najpierw oczywiste spostrzeżenia:

- ▶ najmniejszy krążek znajduje się zawsze na górze któregoś z palików;
- ▶ na górze dwóch pozostałych palików jeden z krążków jest mniejszy od drugiego i tylko ten jeden krążek można przenieść — żaden inny ruch między tymi dwoma palikami nie jest możliwy;
- ▶ z tych dwóch spostrzeżeń wynika, że najmniejszy krążek musi być przenoszony co drugi ruch, a w co drugim ruchu przenoszenie krążka jest jednoznacznie określone — musimy więc jedynie określić, na który palik należy przenieść najmniejszy krążek — szczegóły podajemy już w algorytmie.

Algorytm iteracyjny rozwiązywania łamigłówki Wież Hanoi

Dane: Trzy paliki A, B i C oraz n krążków o różnych średnicach, nanizanych od największego do najmniejszego na palik A. Krążki można przenosić między palikami tylko pojedynczo i nigdy nie można położyć większego na mniejszym.

Wynik: Krążki nanizane na palik B lub C — do uzyskania tego wyniku można wykonywać jedynie dopuszczalne przenoszenia.

Uwaga: Paliki A, B i C traktujemy tak, jakby były ustawione w kółko, zgodnie z ruchem wskazówek zegara, zatem dla każdego palika jest określony następny palik w tym porządku.

Krok 1. Przenieś najmniejszy krążek z palika, na którym się znajduje, na następny palik. Jeśli wszystkie krążki są ułożone na jednym paliku, to zakończ algorytm.

Krok 2. Wykonaj jedyne możliwe przeniesienie krążka, który nie jest najmniejszym krążkiem, i wróć do kroku 1. ■

Ćwiczenie 8.4. Przekonaj się, że kolejność przemieszczania krążków w iteracyjnym algorytmie rozwiązania łamigłówki Wież Hanoi dla $n = 3$ jest dokładnie taka sama, jak w algorytmie rekurencyjnym. ■

Ćwiczenie 8.5. Posłuż się algorytmem iteracyjnym, by rozwiązać łamigłówkę Wież Hanoi dla czterech krążków. Na którym paliku znajdą się wszystkie prawidłowo ułożone krążki? (Zobacz problem 8.1). ■

Rezultaty wykonania kolejnego ćwiczenia będą dobrym dowodem na to, jak rozwiązania rekurencyjne mogą być zwarte i eleganckie w porównaniu z realizacjami algorytmów iteracyjnych dla tego samego problemu.

Ćwiczenie 8.6. Zapisz iteracyjny algorytm rozwiązania łamigłówki wież Hanoi w języku Pascal lub w języku Python. ■

Liczba przeniesień krążków

Na koniec zostawiliśmy pytanie, ile należy wykonać ruchów pojedynczymi krążkami, by rozwiązać łamigłówkę Wież Hanoi z n krążkami — oznaczmy tę liczbę przez h_n . Z dotychczasowych rozważań wiemy już, że: $h_1 = 1$, $h_2 = 3$, $h_3 = 7$. Co przypominają Ci te liczby? Nawet z niewielkim obyciem w dziedzinie algorytmów można zauważyć, że są one o jeden mniejsze od kolejnych potęg liczby 2, mamy więc: $h_1 = 1 = 2^1 - 1$, $h_2 = 3 = 2^2 - 1$, $h_3 = 7 = 2^3 - 1$, i potęga jest równa indeksowi liczby h . Na tej podstawie możemy przypuszczać, że $h_n = 2^n - 1$ dla dowolnej liczby krążków n . Wykażemy, że tak jest rzeczywiście, ale są to trochę trudniejsze rozważania. Ilustrujemy nimi również otrzymywanie zależności rekurencyjnych na podstawie algorytm rekurencyjnego oraz prosty sposób znajdowania rozwiązań takich zależności.

T

Z rekurencyjnego algorytmu rozwiązywania łamigłówki Wież Hanoi wynika następująca **zależność rekurencyjna** między liczbami h_n :

$$h_n = \begin{cases} 1, & n=1 \\ 2h_{n-1} + 1, & n \geq 2 \end{cases} \quad (8.5)$$

Jeśli bowiem $n = 1$, to wykonujemy jedno przeniesienie krążka, a jeśli $n > 1$, to stosujemy rekurencyjny krok algorytmu, w którym dwa razy przenosimy tym samym algorytmem $n - 1$ krążków (wykonując przy tym h_{n-1} przeniesień krążków w obu przypadkach) i raz największy krążek.

W jaki sposób na podstawie zależności rekurencyjnej (8.5) można znaleźć jawną postać, czyli wartości liczb h_n ? W przypadku tej zależności jest to dość proste, gdyż możemy zastosować **metodę wstawiania**. Polega ona na tym, że wielkość stojącą po prawej stronie zależności rekurencyjnej można również wyrazić przez tę samą zależność. Zatem dla $n - 1$ otrzymujemy z zależności (8.5) następującą równość: $h_{n-1} = 2h_{n-2} + 1$, i po wstawieniu jej do wzoru (8.5) mamy:

$$h_n = 2h_{n-1} + 1 = 2(2h_{n-2} + 1) + 1 = 2^2h_{n-2} + 2 + 1 \quad (8.6)$$

Wykonajmy jeszcze jeden krok, by zauważyć pewną regularność. Zależność (8.5) dla $n - 2$ ma postać $h_{n-2} = 2h_{n-3} + 1$, i po wstawieniu jej do wzoru (8.6) otrzymujemy:

$$h_n = 2^2h_{n-2} + 2 + 1 = 2^2(2h_{n-3} + 1) + 2 + 1 = 2^3h_{n-3} + 2^2 + 2 + 1.$$

Takie wstawianie kontynuujemy aż do otrzymania h_1 po prawej stronie znaku równości, wtedy bowiem możemy przerwać rekurencyjne zastępowanie według drugiej części wzoru (8.5) i zastąpić h_1 liczbą 1. Następuje to po $n - 1$ wstawieniach. Wtedy wyrażenie na h_n przyjmuje postać:

$$\begin{aligned} h_n &= 2^{n-1}h_1 + 2^{n-2} + 2^{n-3} + \dots + 2^2 + 2 + 1 = \\ &= 2^{n-1} + 2^{n-2} + \dots + 2^2 + 2 + 1 \quad (8.7) \end{aligned}$$

Wartość sumy po prawej stronie ostatniej równości już wystąpiła w tej książce (zobacz punkt 7.1). Jest to liczba, która w rozwinięciu binarnym ma n jedynek. Następna liczba, czyli większa o 1, ma w rozwinięciu binarnym jedynekę na $n + 1$ pozycji, jest więc równa 2^n . Stąd otrzymujemy ostatecznie:

$$h_n = 2^n - 1 \text{ dla } n \geq 0$$

tak, jak przewidywaliśmy. ■

8.2.2. Liczby Fibonacciego

Jedne z najpopularniejszych liczb występujących w informatyce są związane z pytaniem, jakie zawarł Leonardo Fibonacci w swojej książce *Liber abaci*

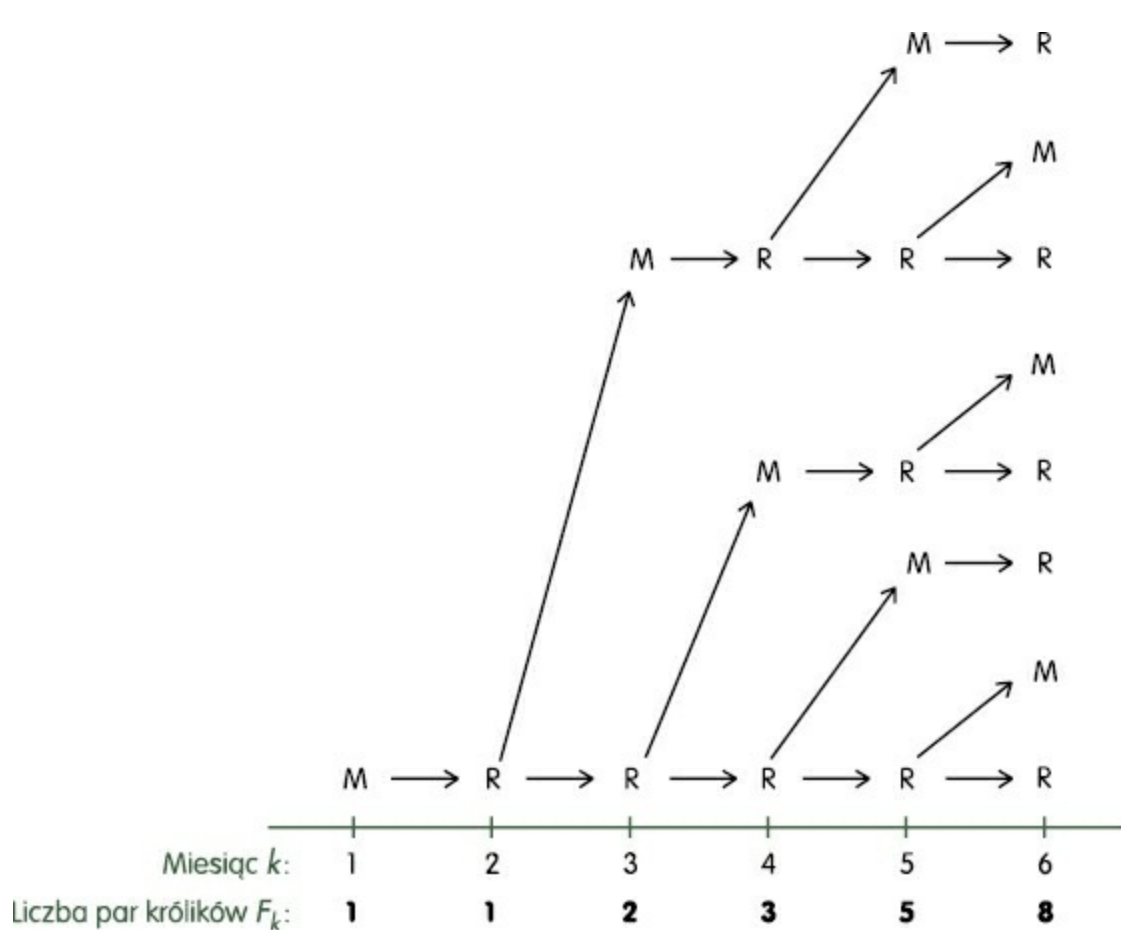
opublikowanej w 1202 roku, a dotyczącym szybkości rozmnażania się królików.

Na początku mamy parę nowo narodzonych królików i o każdej parze królików zakładamy, że:

- ▶ nowa para staje się płodna po miesiącu życia;
- ▶ każda płodna para rodzi jedną parę nowych królików w miesiącu;
- ▶ króliki nigdy nie umierają.

Oryginalne pytanie Fibonacciego brzmiało: ile będzie par królików po upływie roku? Najczęściej pyta się, ile będzie par królików po upływie k miesięcy — oznaczamy tę liczbę przez F_k i nazywamy **liczbą Fibonacciego**. Na rysunku 8.3 przedstawiamy schemat rozrastania się stada królików w ciągu kilku początkowych miesięcy. W pierwszym i drugim miesiącu mamy tylko jedną parę, z tym że w drugim miesiącu może ona dać już parę młodych. Zatem w trzecim miesiącu są już dwie pary, przy czym tylko ta starsza może dalej rodzić młode. Stąd w czwartym miesiącu są już trzy pary, z których dwie, a więc tyle, ile było już w poprzednim miesiącu, mogą rodzić. Czyli w następnym miesiącu mamy te trzy pary i dwie pary młodych, razem pięć par. I tak dalej. Otrzymujemy więc ciąg liczb: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 itd.

Nazwisko Fibonacci pochodzi od brzmienia *Filius Bonacci* (syn Bonacciego). Fibonacci studiował księgi al-Chorezmiego, a liczby nazwane jego nazwiskiem zyskały dużą popularność w XIX wieku, dzięki użyciu ich w badaniu efektywności algorytmu Euklidesa (G. Lamé) — zobacz zadanie 7.1 — oraz w dowodzie, że liczba Mersenne’a dla $p = 127$ jest pierwsza (É. Lucas).



Rysunek 8.3. Schemat rozrastania się stada królików w oryginalnym pytaniu Fibonacciego. M oznacza parę młodych królików, a R — parę dorosłych, czyli rozmnażających się królików

Z tego przykładu oraz z warunków rozmnażania się królików wnioskujemy, że w kolejnym miesiącu liczba par królików będzie równa liczbie par z poprzedniego miesiąca, gdyż króliki nie wymierają, plus liczba par nowo narodzonych królików, a tych jest tyle, ile było par dwa miesiące wcześniej. Zatem kolejna liczba Fibonacciego jest sumą dwóch poprzednich. Stosując oznaczenie na liczbę par królików w danym miesiącu, ten wniosek można zapisać w następującej postaci: $F_k = F_{k-1} + F_{k-2}$, gdzie k jest równe przynajmniej 3, aby można się było odwoływać do poprzednich miesięcy. Musimy zatem wartości dwóch pierwszych liczb Fibonacciego zdefiniować osobno i wprost, nie odwołując się do żadnych innych wartości tego ciągu. Z tych rozważań wynika więc następująca postać liczb Fibonacciego:

$$F_k = \begin{cases} 1, & k=1,2 \\ F_{k-1} + F_{k-2} & k \geq 3 \end{cases} \quad (8.8)$$

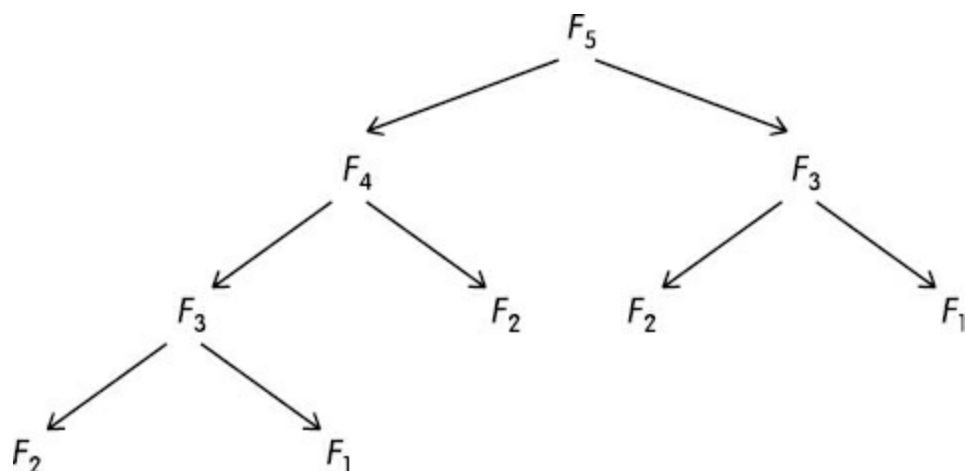
Czy uwierzysz, że w jawnej postaci liczbę Fibonacciego dla dowolnego k wyraża się następującym wzorem:

$$F_k = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right]$$

Wzór (8.8) ma postać **zależności rekurencyjnej** i można go zastosować do obliczania wartości dowolnej liczby Fibonacciego.

Ćwiczenie 8.7. Oblicz wartość liczby F_5 wprost z zależności (8.8). ■

Wykonując to ćwiczenie, można zauważyć, że obliczenia dyktowane wzorem (8.8) nie przebiegają w takiej samej kolejności, jak na rysunku 8.3, zaczynamy je bowiem jakby „od końca”: chcemy obliczyć wartość liczby F_5 , ale nie mamy dwóch poprzednich wartości: F_4 i F_3 . Musimy więc je wyznaczyć, posługując się... tym samym wzorem. Na rysunku 8.4 jest przedstawiony schemat rekurencyjnych odwołań do zależności (8.8) w trakcie obliczania wartości F_5 . Można zauważyć pewną rozrzutność, polegającą na tym, że kilka razy odwołujemy się do tych samych wartości: dwa razy do F_3 , trzy razy do F_2 i dwa razy do F_1 . Te odwołania są rzeczywiście wykonywane niezależnie jedno od drugiego w tym sensie, że na przykład z wartości F_3 obliczonej w lewym poddrzewie nie korzystamy, gdy potrzebna jest nam ta sama wartość w prawym poddrzewie.



Rysunek 8.4. Schemat rekurencyjnych odwołań do wzoru (8.8) w trakcie obliczania z tego wzoru wartości liczby F_5

Niezadowoleni ze sposobu obliczania wartości liczb Fibonacciego, dyktowanego wzorem rekurencyjnym (8.8), wróćmy do sposobu obliczania tych wartości, który wynika wprost z definicji tych liczb, a który wykorzystaliśmy, by utworzyć ilustrację na rysunku 8.4. Ostatecznie z samej natury pytania Fibonacciego wynika, że aby w piątym miesiącu były jakieś króliki, to muszą być już w czwartym, a więc i w trzecim, drugim i pierwszym. A więc liczby Fibonacciego powinno się dać obliczać, zaczynając od najmniejszej. I tak rzeczywiście jest. Zapiszmy od razu ten algorytm w sposób ścisły.

Algorytm iteracyjnego wyznaczania liczb Fibonacciego

Dana: Liczba naturalna k równa przynajmniej 1.

Wynik: Wartość liczby Fibonacciego F_k .

Krok 1. Jeśli $k = 1$ lub $k = 2$, to przyjmij $F_k = 1$ i zakończ algorytm.

Krok 2. Przyjmij $Fib1 := 1$ oraz $Fib2 := 1$. { $Fib1$ oznacza liczbę par królików w poprzednim miesiącu, a $Fib2$ — w dwa miesiące wcześniej.}

Krok 3. Wykonaj $k - 2$ razy następujące instrukcje przypisania:

$Fib = Fib1 + Fib2$; { Fib jest wartością kolejnej liczby Fibonacciego.}

{Dwie następne instrukcje są przygotowaniem do następnej iteracji tego kroku.}

$Fib2 := Fib1$;

$Fib1 := Fib$.

Krok 4. Wartością F_k jest Fib . ■

Poniżej zamieszczamy opisy funkcji FibRek i FibIter w językach Pascal i Python, które są realizacjami rekurencyjnego i iteracyjnego algorytmu obliczania wartości liczby Fibonacciego F_k .



```
function FibRek(k:integer):integer;
begin
  if k <= 2 then Fibrek := 1
  else FibRek := FibRek(k-1) + FibRek(k-2)
end; {FibRek}
```



```
def FibRek(k):
  if k <= 2:
    return 1
  else:
    return FibRek(k-1) + FibRek(k-2)
```



```
function FibIter(k:integer):integer;
var Fib,Fib1,Fib2:integer;
```

```

begin
  if (k=1) or (k=2) then FibIter := 1
  else begin
    Fib1 := 1; Fib2 := 1;
    while k-2 > 0 do begin
      Fib := Fib1 + Fib2;
      Fib2 := Fib1; Fib1 := Fib;
      k := k - 1
    end; {while}
    FibIter := Fib
  end
end; {FibIter}

```



```

def FibIter(k):
    Fk = 1
    Fk1 = 1
    for i in range(k-2):
        Fk, Fk1 = Fk+Fk1, Fk
    return Fk

```

Zadanie 8.1. Porównaj rekurencyjną zależność definiującą liczby Fibonacciego oraz rekurencyjne zależności z poprzednich punktów tego rozdziału. Na tej podstawie postaraj się określić, co jest źródłem nieefektywności w rekurencyjnym algorytmie obliczania wartości liczby Fibonacciego F_k . ■



Polecamy lekturę rozdziału 6. w książce [Piramidy], poświęconego m.in. związkom liczb Fibonacciego z różnymi tworam natury ożywionej i nieożywionej. Omówiono tam także różne algorytmy obliczania wartości liczb Fibonacciego i porównano ich efektywność. ■

8.3. Zadania i problemy

Zadanie 8.2. Dla sprawdzenia, że rozumiesz podobieństwa i różnice między iteracją i rekurencją, zapisz sumę $s = \sum_{i=1}^n a_i$ w postaci zależności rekurencyjnej i

opisz algorytm obliczania wartości s według tej zależności oraz zapisz go w języku Pascal lub w języku Python. Sprawdź swoje programy na komputerze.

Zadanie 8.3. Podaj rekurencyjną definicję funkcji **silnia**:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1.$$

Zapisz w języku Pascal lub/i w języku Python algorytm obliczania wartości funkcji silnia według jej rekurencyjnej definicji.

Zadanie 8.4. Wykonaj zadanie podobne do dwóch poprzednich, ale tym razem dla obliczania najmniejszego elementu w ciągu: $\min = \min\{a_i : 1 \leq i \leq n\}$.

Problem 8.1. Kontynuując ćwiczenie 8.5 — czy potrafisz podać, od czego zależy, że w algorytmie iteracyjnym rozwiązywania łamigłówki Wież Hanoi krążki są przenoszone na palik B lub C? Czym to można uzasadnić?

Zadanie 8.5. Porównaj iteracyjny i rekurencyjny algorytm rozwiązywania łamigłówki Wież Hanoi, wykonując obliczenia dla n równego 3 i 4 oraz porównaj sekwencje przemieszczeń krążków otrzymane tymi algorytmami.

Zadanie 8.6. Przyjmij, że grupa mnichów w Tybecie zaczęła rozwiązywać łamigłóvkę Wież Hanoi z $n = 64$ krążkami na początku naszej ery, a więc 2016 lat temu, i pracuje z prędkością komputera średniej mocy, przenosząc 100 miliardów, czyli 10^9 , pojedynczych krążków na sekundę! Według legendy, po rozwiązaniu przez nich tej łamigłówki ma nastąpić koniec świata. Oblicz, kiedy to nastąpi.

Zadanie 8.7. Profesor S. bardzo chaotycznie chodzi po schodach i czasem pokonuje dwa schody, a czasem tylko jeden. Na ile sposobów profesor S. może wejść do swojego gabinetu, mieszczącego się na półpiętrze, które dzieli od parteru 10 schodów? Wyprowadź ogólną zależność na liczbę różnych sposobów pokonania n schodów przez profesora S.

Zadanie 8.8. Sprawdź na przykładzie kilku początkowych liczb Fibonacciego (dla $n = 1, 2, 3$), że rzeczywiście słuszny jest dla nich wzór podany w drugiej ramce w punkcie 8.2.2.

Miałeś okazję dowiedzieć się, że:

- rozwiązywanie problemów polega często na **korzystaniu z istniejących rozwiązań** lub z rozwiązania problemu... właśnie rozwiązywanego — w tym drugim przypadku możemy się posłużyć **rekurencją**;
- **iteracja może być również zapisana jako rekurencja**;
- **rekurencję** w algorytmie można na ogół **zastąpić iteracją**;
- stosowanie **zależności rekurencyjnych**, chociaż na ogół są one bardziej eleganckim i zwartym zapisem, może **prowadzić do**

zmniejszenia efektywności obliczeń;

oraz poznać:

- ▶ sposób **zapisywania iteracji w postaci rekurencji;**
- ▶ rekurencyjny i iteracyjny sposób rozwiązywania łamigłówki **Wież Hanoi;**
- ▶ rekurencyjny i iteracyjny sposób wyznaczania wartości **liczb Fibonacciego;**
- ▶ sposoby **zamiany rekurencji na iterację;**
- ▶ **komputerową realizację procedur rekurencyjnych**, czyli procedur odwołujących się do siebie samych.

Rozdział 9. Dziel i zwyciężaj

Tu poznasz bliżej **zasadę dziel i zwyciężaj** oraz jej zastosowanie w rozwiązaniach takich problemów, jak:

- ▶ **znajdowanie w ciągu największego i najmniejszego elementu jednocześnie** — ponownie, ale tym razem za pomocą algorytmu rekurencyjnego;
- ▶ **przeszukiwanie ciągu uporządkowanego** metodą połowienia;
- ▶ **przeszukiwanie interpolacyjne** katalogów i słowników;
- ▶ **znajdowanie miejsca zerowego funkcji** w danym przedziale przez jego sukcesywne połowienie.

W punkcie 5.5 rozważaliśmy problem znajdowania w zbiorze największego i najmniejszego elementu jednocześnie i podaliśmy algorytm MaxMin, w którym można wyróżnić trzy etapy:

1. Podziel problem na podproblemy.
2. Znajdź rozwiązania podproblemów.
3. Połącz rozwiązania podproblemów w rozwiązanie głównego problemu.

Pierwszy etap polega na podziale zbioru danych na dwa podzbiory: kandydatów na maksimum i kandydatów na minimum, drugi — na znajdowaniu maksimum i minimum w zbiorach kandydatów, a etap trzeci jest jedynie wyprowadzeniem rozwiązania składającego się z dwóch liczb, otrzymanych jako rozwiązania dwóch podproblemów.

Podkreślaliśmy już wielokrotnie, że naturalnym podejściem w rozwiązywaniu problemów powinno być dążenie do wykorzystania znanych rozwiązań problemów — miejscem ku temu w powyższym schemacie jest etap drugi. W przypadku problemu MaxMin korzystamy na tym etapie ze znanych algorytmów znajdowania największego i najmniejszego elementu w zbiorze.

W tym rozdziale rozważamy kolejne problemy, których rozwiązanie jest budowane według przytoczonego schematu. Określimy najpierw, jakie dodatkowe własności powinny mieć podproblemy rozwiązywane w drugim etapie, aby skonstruowany algorytm miał dobre własności i był efektywny:

- 1a.** Problem jest dzielony na takie same lub bardzo podobne podproblemy.
- 1b.** Liczba podproblemów wynosi co najmniej 2.
- 1c.** Podproblemy są rozwiązywane na podzbiorach zbioru danych, w których liczba elementów jest niemal jednakowa i stanowi stałą część (np. połowę) całego zbioru danych rozwiązywanego problemu.

Problem MaxMin oraz algorytm jego rozwiązywania podany w punkcie 5.5 spełniają te trzy dodatkowe warunki: wydzielone podproblemy (znalezienia maksimum i znalezienia minimum) mają bardzo podobną naturę, ich liczba wynosi dwa i odnoszą się do podzbiorów danych, które mają podobną wielkość, równą niemal połowie zbioru występującego w całym problemie (jeśli liczba elementów w zbiorze danych jest parzysta, to oba problemy działają dokładnie na połowie elementów spośród danych).

Metoda rozwiązywania problemu według powyższego schematu nazywa się **metodą dziel i zwyciężaj** — tę nazwę wprowadziliśmy już w punkcie 5.5. W najczęściej stosowanej wersji tej metody, podproblemy, na które jest rozkładany problem, są tymi samymi problemami, ale dla mniejszego zbioru danych, można więc rekurencyjnie zastosować do nich tę samą metodę rozwiązywania.

W tym rozdziale stosujemy metodę dziel i zwyciężaj do rozwiązania kilku problemów. Zaczniemy od powtórnego spojrzenia na problem znajdowania jednocześnie największego i najmniejszego elementu w zbiorze i podamy algorytm, w którym rekurencyjnie jest rozwiązywany ten sam problem na podzbiorach o połowę mniejszych. Następnie opisujemy problemy poszukiwania przez połowienie. W przypadku tych problemów zasada dziel i zwyciężaj jest w pewnym sensie nawet silniejsza — przestrzeń przeszukiwania jest dzielona na dwie części, ale dalsze obliczenia przebiegają tylko w jednej z nich, gdyż poszukiwane rozwiązanie znajduje się tylko w jednej z części. Metodę dziel i zwyciężaj stosujemy również w następnym rozdziale do wyprowadzenia szybkich algorytmów porządkowania ciągów liczb.

9.1. Rekurencyjne znajdowanie największego i najmniejszego elementu

Omówimy teraz inny algorytm rozwiązywania problemu MaxMin, który jest również realizacją metody dziel i zwyciężaj, ale przy podziale problemu na podproblemy nie uwzględnia się w nim specyfiki problemu, tylko dzieli zbiór danych na dwie niemal równe części i rozwiązuje w nich ten sam problem. Wyniki uzyskane w podproblemach, czyli największe i najmniejsze w nich elementy, są następnie porównywane, by określić największy i najmniejszy element w całym zbiorze. Jest to więc typowy algorytm rekurencyjny. Jako szczególny przypadek, czyli warunek zakończenia rekurencji, wyróżniamy ten sam problem na zbiorach złożonych z jednego lub z dwóch elementów.

Podobnie jak w opisie rekurencyjnego algorytmu rozwiązywania łamigłówki Wież Hanoi, w nazwie tego algorytmu umieszczamy wykaz parametrów, dla których algorytm znajduje rozwiązanie. Dzięki temu wewnątrz algorytmu możemy odwołać się do... tego samego algorytmu, z odpowiednio dobranym wykazem parametrów.

Algorytm MinMaxRek (Z , \min , \max)

Dane: Zbiór liczb Z .

Wyniki: \min i \max , odpowiednio najmniejszy i największy element w zbiorze Z .

Krok 1. Jeśli zbiór Z składa się z jednego elementu, to przypisz jego wartość zarówno \min , jak i \max .

Jeśli zbiór Z składa się z dwóch elementów, to wartość mniejszego z nich (lub równego drugiemu) przypisz \min , a wartość większego z nich — \max .

Krok 2. W przeciwnym razie:

2a. Podziel zbiór Z na dwa podzbiory Z_1 i Z_2 o tej samej lub niemal tej samej liczbie elementów.

2b. Wykonaj ten sam algorytm dla (Z_1, \min_1, \max_1) .

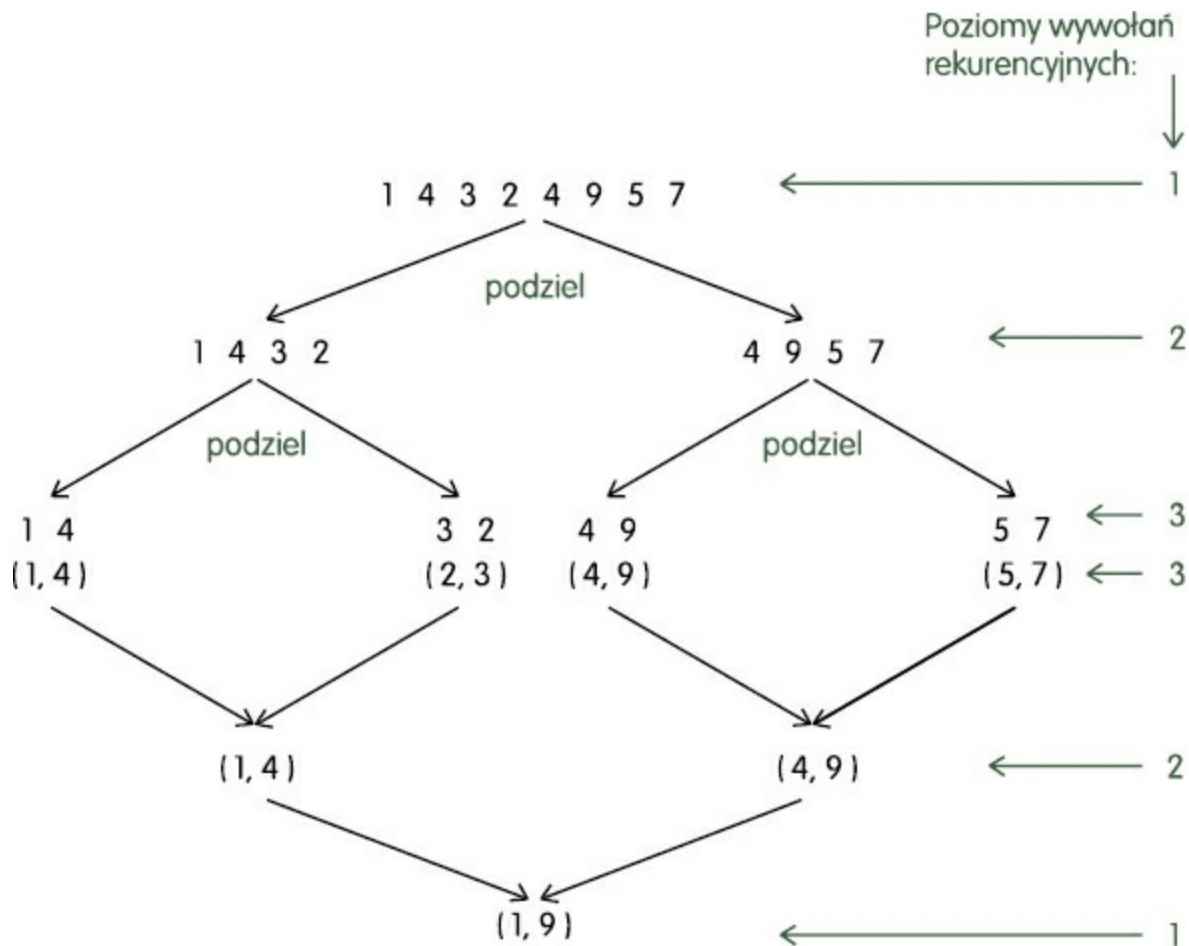
2c. Wykonaj ten sam algorytm dla (Z_2, \min_2, \max_2) .

2d. Wartość mniejszej z liczb \min_1 i \min_2 (lub równej drugiej) przypisz \min , wartość większej z liczb \max_1 i \max_2 (lub równej drugiej) przypisz \max . ■

Na rysunku 9.1 jest przedstawiony przykładowy przebieg algorytmu MinMaxRek dla danych z rysunku 5.6.

Ćwiczenie 9.1. Wykonaj algorytm MinMaxRek dla zbioru pierwszych siedmiu liczb z przykładu pokazanego na rysunku 9.1. Ile wykonałeś porównań? ■

Poniżej przedstawiamy realizacje algorytmu MinMaxRek w językach Pascal i Python. Zbiór Z jest zapisany w tablicy $x[i..j]$ liczb naturalnych. Wyznaczane rozwiązanie ma ogólniejszą postać niż w algorytmie powyżej, gdyż zamiast samych elementów, najmniejszego i największego, znajdowane są ich indeksy w przeszukiwanym ciągu.



Rysunek 9.1. Przykład działania algorytmu MinMaxRek na zbiorze danych z rysunku 5.5. Para (x, y) oznacza (\min, \max) obliczone w wyniku zamknięcia wywołania rekurencyjnego



```

procedure MinMaxRek(i,j:integer; var min,max:integer);
  var k,min1,min2,max1,max2:integer;
begin
  if i = j then
    begin min := i; max := i end
  else
    if i + 1 = j then
      if x[i] >= x[j] then
        begin min := j; max := i end
      else begin min := i; max := j end
    else begin
      k := (i + j) div 2;
      MinMaxRek(i,k,min1,max1);
      MinMaxRek(k+1,j,min2,max2);
    end
  end

```

```

if x[min1] <= x[min2] then min := min1
else min := min2;
if x[max1] >= x[max2] then max := max1
else max := max2
end
end; {MinMaxRek}

```



```

def MinMaxRek(i,j):
    if i == j:
        Min,Max = i,j
    else:
        if i + 1 == j:
            if x[i] >= x[j]:
                Min,Max = j,i
            else:
                Min,Max = i,j
        else:
            k = (i + j) // 2
            min1,max1 = MinMaxRek(i,k)
            min2,max2 = MinMaxRek(k+1,j)
            if x[min1] <= x[min2]:
                Min = min1
            else:
                Min = min2
            if x[max1] >= x[max2]:
                Max = max1
            else:
                Max = max2
    return Min,Max

```

Złożoność

W punkcie 5.5 obliczyliśmy dość łatwo, że w przedstawionym tam algorytmie

MaxMin są wykonywane $\lceil 3n/2 \rceil - 2$ porównania na zbiorze złożonym z n

elementów i stwierdziliśmy, że jest to najmniejsza możliwa liczba porównań w algorytmie dla tego problemu, czyli jest on optymalny. A jaka jest złożoność rekurencyjnego algorytmu rozwiązywania tego problemu? Odpowiedź na to

pytanie w ogólnym przypadku nie jest już taka prosta; wyprowadzimy wzór na liczbę porównań jedynie dla n o szczególnej postaci.

T

Jeśli $n = 1$, to jedyny element zbioru jest jednocześnie jego największym i najmniejszym elementem. Jeśli $n = 2$, to wykonujemy jedno porównanie, by stwierdzić, który z elementów jest większy, a który mniejszy. Jeśli $n > 2$, to zbiór elementów jest dzielony na dwa prawie równoliczne podzbiory i rekurencyjnie jest wywoływany dla nich ten sam algorytm. Można by w tym momencie napisać zależność rekurencyjną dla liczby porównań (jak to zrobiliśmy w przypadku algorytmu rozwiązywania łamigłówek Wież Hanoi), ale nie wiadomo, ile elementów jest w każdym z podzbiorów. Jeśli bowiem n jest liczbą parzystą, to każdy podzbiór zawiera dokładnie połowę elementów, czyli $n/2$. Jeśli natomiast n jest nieparzyste, to jeden z podzbiorów zawiera o jeden element więcej niż drugi podzbiór.

Sprawdź na przykładach, że równość $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$ jest prawdziwa dla każdej liczby naturalnej n oraz że liczby $\lfloor n/2 \rfloor$ i $\lceil n/2 \rceil$ różnią się od siebie o co najwyżej

1. Zatem w algorytmie dziel i zwyciężaj zbiór złożony z n elementów można zawsze podzielić na dwa podzbiory o takich licznosciach, które są sobie bliskie. Niestety obliczenia, w których występują funkcje podłogi i powały, są dość złożone, dlatego ich tutaj nie przytaczamy.

Nie potrafimy jednak zapisać tego w sposób elementarny. Dlatego w dalszych rozważaniach dotyczących złożoności algorytmu MaxMinRek zakładamy, że zbiór Z , w którym szukamy największego i najmniejszego elementu, ma tyle elementów, że zawsze można go podzielić na dwie równe części, i podobnie — wszystkie podzbiory otrzymywane z podziałów można również podzielić na dwa równoliczne podzbiory. Dla jakiego n jest to możliwe?

Ćwiczenie 9.2. Sprawdź najpierw na przykładach, że jeśli n nie jest potęgą liczby 2, to dzieląc n wielokrotnie przez 2 otrzymamy w końcu liczbę nieparzystą większą od 1. Następnie wykaż to dla dowolnej liczby n . ■

Na podstawie rozwiązania tego ćwiczenia, jeśli zbiór i jego podzbiory, otrzymywane z podziałów, mają dawać się dzielić na dwa równoliczne podzbiory, to musimy przyjąć, że $n = 2^k$ dla pewnego k . Dla takiego n możemy zapisać teraz zależność rekurencyjną na liczbę porównań wykonywanych w algorytmie MaxMinRek — oznaczmy tę liczbę przez $g(n)$.

$$g_n = \begin{cases} 0, & n=1 \\ 1, & n=2 \\ 2g(n/2)+2, & n>2 \end{cases} \quad (9.1)$$

Zależność (9.1) wynika wprost z postaci algorytmu: gdy $n = 1$, nie wykonujemy żadnego porównania między elementami zbioru Z , gdy $n = 2$ — wykonujemy jedno, gdy $n > 2$, najpierw rozwiązujemy dwa takie same problemy na podzbiorach o połowę mniejszych — liczba porównań wykonywanych w rozwiązaniach tych podproblemów wynosi $g(n/2)$ — a następnie wykonujemy dwa porównania, porównując ze sobą największe i najmniejsze elementy z tych podzbiorów. Ponieważ n jest potęgą liczby 2, wielokrotne dzielenie n na pół zawsze daje liczbę całkowitą, a więc zależność (9.1) jest poprawna dla każdego poziomego rekurencyjnego wywołania algorytmu MaxMinRek. Tę zależność można rozwiązać (podobnie, jak rozwiązyaliśmy zależność dla Wieży Hanoi), stosując metodę wstawiania — zobacz punkt 8.2.1. Jedyna różnica i trudność polega na tym, że we wzorze (8.5) jest odwołanie do tej samej wielkości z indeksem o jeden mniejszym, a w zależności (9.1) jest odwołanie do funkcji g z argumentem o połowę mniejszym. Spróbuj jednak swoich sił.

Gdy wstawimy $n/2$ w miejsce n w zależności (9.1), otrzymamy: $g(n/2) = 2g(n/4) + 2$, i po wstawieniu do zależności (9.1) mamy:

$$g(n) = 2g(n/2) + 2 = 2(2g(n/4) + 2) + 2 = 2^2g(n/2^2) + 2^2 + 2$$

Jeśli $n/2^2$ nie równa się 2, to możemy ponownie skorzystać z zależności (9.1), by zastąpić po prawej stronie $g(n/2^2)$, i otrzymamy:

$$g(n) = 2^2g(n/2^2) + 2^2 + 2 = 2^2(2g(n/2^3) + 2) + 2^2 + 2 = 2^3g(n/2^3) + 2^3 + 2^2 + 2.$$

To wstawianie kończymy, gdy n podzielone przez potęgę liczby 2 daje w wyniku 2 — wtedy bowiem korzystamy z drugiej równości w zależności (9.1), która jest warunkiem zakończenia rekurencji. Następuje to po $k - 1$ krokach, gdyż, jak założyliśmy, $n = 2k$. Wtedy otrzymujemy następujący ciąg równości:

$$\begin{aligned}
g(n) &= 2^{k-1} g(n/2^{k-1}) + 2^{k-1} + \dots + 2^2 + 2 = 2^{k-1} g(2) + 2^{k-1} + \dots + 2^2 + 2 = \\
&= 2^{k-1} + 2^{k-1} + \dots + 2^2 + 2 = \\
&= 2 \cdot 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2 = 2^k + 2^{k-2} + \dots + 2^2 + 2 = \\
&= 2^k + (2^{k-2} + \dots + 2^2 + 2 + 1) - 1 = 2^k + (2^{k-1} - 1) - 1 = \\
&= 3 \cdot 2^{k-1} - 2 = \frac{3}{2} 2^k - 2 = \\
&= \frac{3}{2} n - 2.
\end{aligned}$$

zakończony identycznym wzorem, jak w przypadku algorytmu MaxMin. Pamiętajmy jednak, że ten wzór jest prawdziwy dla $n = 2^k$. Dla n nie będących potęgą liczby 2 możemy jedynie przypuszczać, że liczba porównań jest w przybliżeniu taka sama, ale dokładne sformułowanie tego stwierdzenia i jego uzasadnienie wykracza znacznie poza zakres tej książki.



Z dokładnym określeniem wartości funkcji $g(n)$ dla wszystkich wartości n można się zapoznać w książkach [BanKrecz], [Knuth-2]. ■

Wykonaliśmy powyższe obliczenia, gdyż są one charakterystyczne dla obliczeń złożoności większości algorytmów zbudowanych na zasadzie dziel i zwyciężaj, w których problem jest dzielony na dwa takie same podproblemy o prawie jednakowej liczbie elementów (zobacz np. algorytm porządkowania przez scalanie przedstawiony w punkcie 10.2.2).

9.2. Przeszukiwanie binarne, czyli przez połowienie

W punkcie 5.1.2 rozważaliśmy poszukiwanie elementu w zbiorze, o którym nic nie wiemy, na przykład, czy elementy są w nim uporządkowane. W takim przypadku, aby mieć pewność, że nie pominęliśmy żadnego elementu zbioru, przeszukujemy go element po elemencie, tj. zakładamy, że elementy zbioru zostały ustawione w ciągu i przeszukujemy ten ciąg od któregoś z jego końców. Taka metoda nazywa się **przeszukiwaniem liniowym**.

W wielu sytuacjach praktycznych zbiory, w których poszukujemy elementów, mają jednak pewną strukturę — są na przykład uporządkowane. Powinniśmy więc umieć wykorzystać w algorytmach te dodatkowe informacje o zbiorze.

Zastosujemy teraz metodę dziel i zwyciężaj do przeszukiwania zbiorów uporządkowanych, mając nadzieję, że przyniesie to nam zwycięstwo nad metodą przeszukiwania liniowego.

Ćwiczenie 9.3. Przypuśćmy, że archiwum w Twojej szkole nie zostało jeszcze skomputeryzowane, ale panuje w nim jakiś „rozsądny” porządek. Po kilku latach od chwili opuszczenia przez Ciebie szkoły masz znaleźć teczkę ze swoimi dokumentami. Jakbyś jej szukał? ■

Ćwiczenie 9.4. Otrzymałeś wydrukowaną listę nazw plików znajdujących się w jakimś folderze. Chciałbyś sprawdzić, czy znajduje się wśród nich plik, którego szukasz, o podanej nazwie, rozszerzeniu i dacie utworzenia. Jaką obrałbyś drogę poszukiwania? Rozważ różne możliwe sposoby utworzenia listy plików. ■

Te dwa ćwiczenia nie dostarczą Ci ogólnej metody przeszukiwania różnych zbiorów, które prawdopodobnie zostały w jakiś sposób uporządkowane, ale mają zwrócić Twoją uwagę na złożoność operacji przeszukiwania w rzeczywistych sytuacjach.

W dalszych rozważaniach w tym punkcie będziemy zakładać, że elementy zbioru, który przeszukujemy, są liczbami i zostały uprzednio uporządkowane. W tej książce dużo mówimy o porządkowaniu zbiorów i o wpływie uporządkowania zbioru na efektywność wykonywania na nim różnych operacji. Również przeszukiwanie liniowe może być znacznie przyspieszone, gdy ciąg jest uporządkowany i gdy korzystamy z tego w algorytmie (zobacz problem 5.2). Możliwy jest jeszcze bardziej efektywny algorytm przeszukiwania zbioru uporządkowanego. Zaczniemy od zabawy.

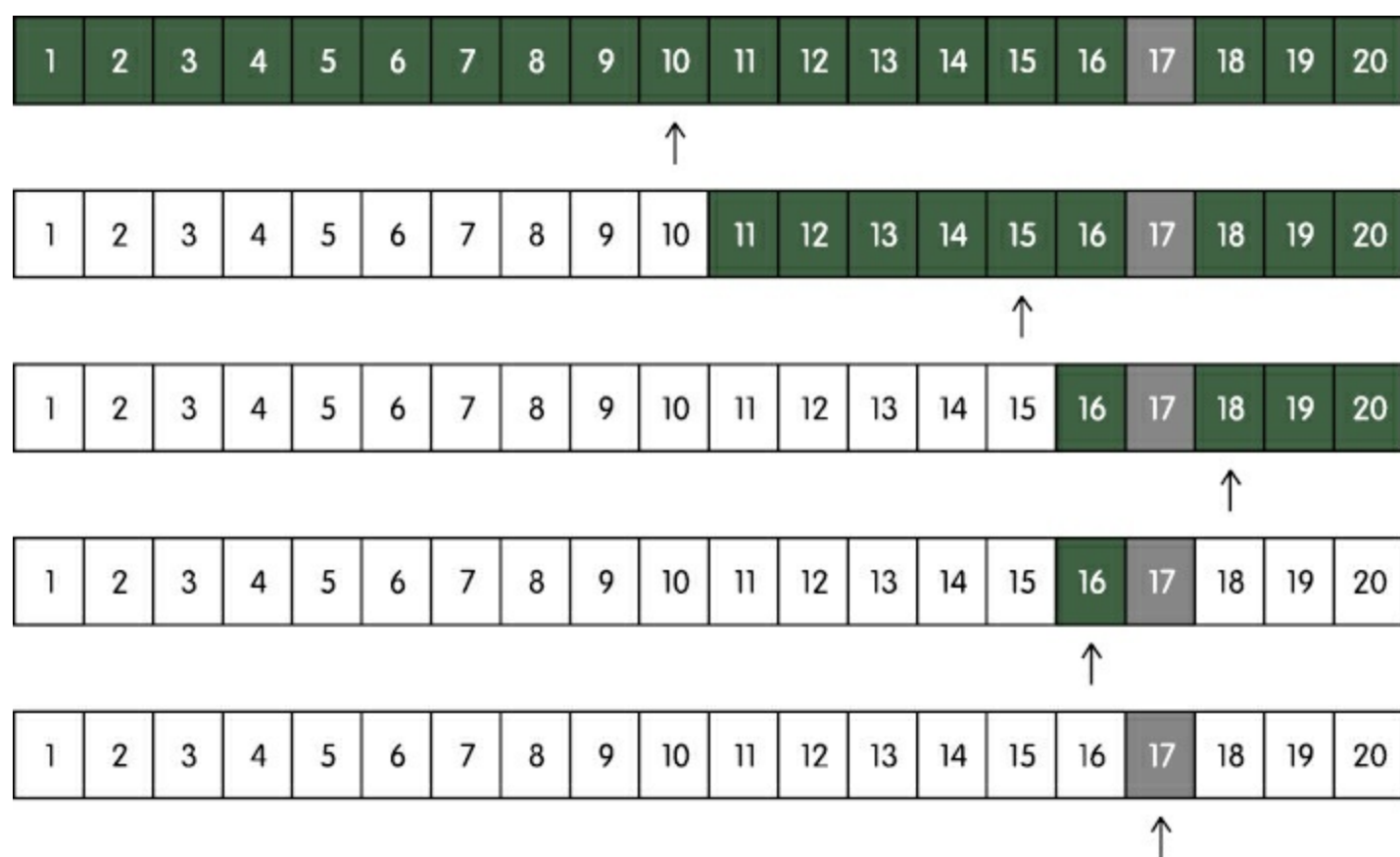
Zabawa w zgadywanie liczb

Ćwiczenie 9.5. 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 pozostałoby 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 $[51, 100]$.

Na rysunku 9.2 jest przedstawiony przebieg poszukiwania liczby 17 wśród dwudziestu początkowych liczb naturalnych, w którym 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 mniejsza. Podciąg liczb, który pozostał do przeszukania, jest oznaczony na zielono, by zwrócić Twoją uwagę na fakt, że w każdym kroku odgadywania ten ciąg jest redukowany przynajmniej o połowę. Taką metodę nazywa się **przeszukiwaniem binarnym**.



Rysunek 9.2. Przykład zastosowania binarnej metody przeszukiwania do znalezienia liczby 17 w ciągu złożonym z dwudziestu początkowych liczb naturalnych. Na zielono oznaczono podciąg, w którym znajduje się poszukiwana liczba, a strzałka wskazuje wybraną w nim liczbę

Algorytm przeszukiwania

Opiszemy teraz algorytm binarnego przeszukiwania dla trochę ogólniejszej sytuacji niż w zabawie w odgadywanie liczb. Po pierwsze zauważmy, że w podanej wyżej metodzie nie mają znaczenia wartości elementów, wśród których prowadzimy poszukiwania, tak długo, jak długo są uporządkowane. Musimy mieć jedynie pewność, że po porównaniu wybranej liczby z poszukiwaną i po wybraniu połowy zbioru poszukiwana liczba znajduje się w wybranej połowie, a do tego wystarczy, by przeszukiwane elementy były uporządkowane, nie muszą być one koniecznie kolejnymi liczbami. Mogą wśród nich być również takie same liczby — wtedy oczywiście zajmują one miejsca obok siebie. Po drugie poszukiwana liczba nie musi znajdować się wśród przeszukiwanych — wtedy naszą odpowiedzią będzie jakaś specjalnie wybrana liczba, np. -1 .

Przyjmijmy, że przeszukiwany ciąg liczb jest umieszczony w tablicy $a[k..l]$ (w języku Python będzie to lista). Załóżmy dodatkowo, że wartość poszukiwanego elementu y mieści się w przedziale wartości elementów w tej tablicy, czyli $a_k \leq y \leq a_l$. Stosowanie algorytmu binarnego gwarantuje, że w trakcie jego działania (podobnie jak w grze w odgadywanie liczby) przeszukiwany przedział zawiera element y , czyli $a_{\text{lewy}} \leq y \leq a_{\text{prawy}}$. Ta własność oraz to, że długość tego przedziału zmniejsza się w każdej iteracji (zobacz krok 3.), zapewniają poprawność poniższego algorytmu (zobacz punkt 12.2, gdzie ściśle wykazujemy poprawność tego algorytmu).

Algorytm binarnego przeszukiwania

Dane: Uporządkowany ciąg liczb w tablicy $a[k..l]$, gdzie $k \leq l$, tzn. $a_k \leq a_{k+1} \leq \dots \leq a_l$; oraz element y spełniający nierówności $a_k \leq y \leq a_l$.

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

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

Krok 2. Jeśli $\text{lewy} > \text{prawy}$, to przypisz $s := -1$ i zakończ algorytm.

{Elementu y nie ma w przeszukiwanej tablicy.}

Krok 3. $s := (\text{lewy} + \text{prawy}) \text{ div } 2$; Jeśli $a_s = y$, to zakończ algorytm.
{Znaleziono element y w przeszukiwanej tablicy.}

Jeśli $a_s < y$, to $\text{lewy} := s + 1$, w przeciwnym razie $\text{prawy} := s - 1$.

Wróć do kroku 2. ■

Ćwiczenie 9.6. Zastosujmy wspólnie (sprawdzaj obliczenia) powyższy algorytm do przeszukania ciągu liczb znajdującego się w tablicy przedstawionej na rysunku 9.2, ale będziemy szukać elementu $y = 17.5$, którego w niej nie ma. W pierwszej iteracji kroków 2. i 3. otrzymujemy $s := 10$ oraz $\text{lewy} := 11$, w drugiej iteracji: $s := 15$ i $\text{lewy} := 16$, w trzeciej: $s := 18$ i $\text{prawy} := 17$, w czwartej: $s := 16$ i $\text{lewy} := 17$, a w piątej: $s := 17$ i $\text{lewy} := 18$. W następnej iteracji, ponieważ $\text{lewy} > \text{prawy}$, następuje zakończenie algorytmu z wynikiem $s = -1$, oznaczającym, że nie znaleziono w tablicy elementu o wartości 17.5. ■

Jak podaje Donald E. Knuth [Knuth-3], pierwszy opis metody binarnej pojawił się w latach 40., ale dopiero na początku lat 60. podano jej poprawny opis i bezbłędną realizację komputerową.

Poniżej zamieszczamy opisy przeszukiwania binarnego w językach Pascal i Python. Patrz również dalej w tym rozdziale, gdzie omawiamy algorytm binarnego umieszczania. Polecamy również rozważania w punkcie 12.2, gdzie omawiamy poprawność algorytmów binarnego poszukiwania i umieszczania.



```
function PrzeszukiwanieBinarne(a:TablicaIn; k,l:integer; :integer) :integer;
  var Lewy,Prawy,Srodek:integer;
begin
  Lewy := k; Prawy := l;
  while Lewy <= Prawy do begin
    Srodek := (Lewy + Prawy) div 2;
    if a[Srodek] = y then begin
      PrzeszukiwanieBinarne := Srodek; exit
    end;
    if a[Srodek] < y then Lewy := Srodek + 1
    else Prawy := Srodek - 1
  end;
  PrzeszukiwanieBinarne := -1
end; {PrzeszukiwanieBinarne}
```



```
def BinarySearch(y,a,k,l):
  Lewy,Prawy = k,l
  while Lewy <= Prawy:
    Środek =(Lewy + Prawy) // 2
    if a[Środek] == y:
      return Środek
    else:
      if a[Środek] < y:
        Lewy = Środek + 1
      else:
        Prawy = Środek - 1
  return -1
```

Złożoność

Nasuwa się teraz pytanie, ile porównań jest wykonywanych w algorytmie binarnego przeszukiwania. Załóżmy, że poszukiwany element znajduje się w ciągu — ponieważ jeśli go tam nie ma, to jest wykonywana jedna dodatkowa iteracja (przekonaj się o tym). Dla dwudziestu liczb z przykładu podanego na rysunku 9.2 poszukiwaną liczbę znaleźliśmy po czterech porównaniach — piątego nie musieliśmy już wykonywać, gdyż pozostała dokładnie jedna liczba —

poszukiwana.

Ćwiczenie 9.7. Przeprowadź następujący eksperyment obliczeniowy: za poszukiwaną liczbę wybieraj kolejne liczby naturalne 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 ćwiczenia 9.7 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 (zobacz zadanie 7.5). Pojawia się więc tutaj ponownie funkcja odwrotna do potęgowania — logarytm. Dokładniej określimy liczbę kroków w metodzie binarnej po przedstawieniu tej metody w nieco ogólniejszej postaci.

Zbiór n elementowy można połowić co najwyżej tylko $\lceil \log_2 n \rceil$ razy.

Algorytm umieszczania

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

Ćwiczenie 9.8. Przypomnij sobie algorytm porządkowania pięciu liczb opisany w punkcie 4.3, w którym wykonuje się 7 porównań. Odszukaj w nim dwa kroki, w których jest stosowane binarne umieszczanie. Jaka byłaby złożoność tego algorytmu, gdyby zastosowano w nim liniowe umieszczanie? ■

Hugo Steinhaus w swoim *Kalejdoskopie matematycznym* opisał binarne umieszczanie i zastosował je w algorytmie porządkowania (zobacz punkt 10.1), długo uważanym za najszybszy.

Poniższy algorytm dla danego elementu y znajduje miejsce, w które można go wstawić z zachowaniem porządku. Poprawność tego algorytmu uzasadnia się podobnie, jak poprawność algorytmu przeszukiwania. W tym celu wystarczy przyjąć, że umieszczany element nie jest mniejszy od lewego końca przedziału, czyli $y \geq ak$.

Algorytm binarnego umieszczania

Dane: Uporządkowany ciąg liczb w tablicy $a[k..l]$, gdzie $k \leq l$, to znaczy $a_k \leq a_{k+1} \leq \dots \leq a_l$; oraz element y spełniający nierówność $y \geq a_k$.

Wynik: Miejsce dla y w ciągu $a[k..l]$, tzn. największe r takie, że $a_r \leq y \leq a_{r+1}$, jeśli $k \leq r \leq l-1$, lub $r = l$, gdy $a_l \leq y$. {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. $s := \lceil (lewy + prawy) / 2 \rceil$.

Jeśli $a_s \leq y$, to $lewy := s$, w przeciwnym razie $prawy := s - 1$.

Jeśli $lewy = prawy$, to zakończ algorytm — wtedy $r = lewy$, a w przeciwnym razie powtórz krok 2. ■

Ćwiczenie 9.9. Wykonaj algorytm binarnego umieszczania dla danych z ćwiczenia 9.6, czyli dla ciągu złożonego z dwudziestu początkowych liczb naturalnych, począwszy od 1, i dla $y = 17.5$. Porównaj na tym przykładzie działanie obu algorytmów binarnych, przeszukiwania i umieszczania. ■

Ćwiczenie 9.10. Sprawdź na przykładzie poprawność algorytmu binarnego umieszczania, gdy wartość elementu y wykracza poza prawy koniec przedziału, czyli gdy $y \geq a_l$. A jaki otrzymasz wynik, gdy wartość poszukiwanego elementu, wbrew założeniu, wykracza poza lewy koniec przedziału, czyli gdy $y < a_k$? ■

Algorytm binarnego umieszczania jest opisany poniżej w postaci funkcji w języku Pascal. Opis w języku Python pozostawiamy do samodzielnego wykonania. ■



```
function UmieszczanieBinarne(a:TablicaIn; k,l:integer; y:integer) :integer;
var Lewy,Prawy,Srodek,Pom:integer;
begin
  Lewy := k; Prawy := l;
  repeat
    Pom := Lewy + Prawy;
    if Pom mod 2 <> 0 then
      Srodek := (Pom + 1) div 2
    else Srodek := Pom div 2;
    {Srodek jest równy powale z połowy Lewy+Prawy.}
    if a[Srodek] <= y then Lewy := Srodek
```

```

else Prawy := Srodek - 1
until Lewy = Prawy;
UmieszczanieBinarne := Lewy
end; {UmieszczanieBinarne}

```

Złożoność

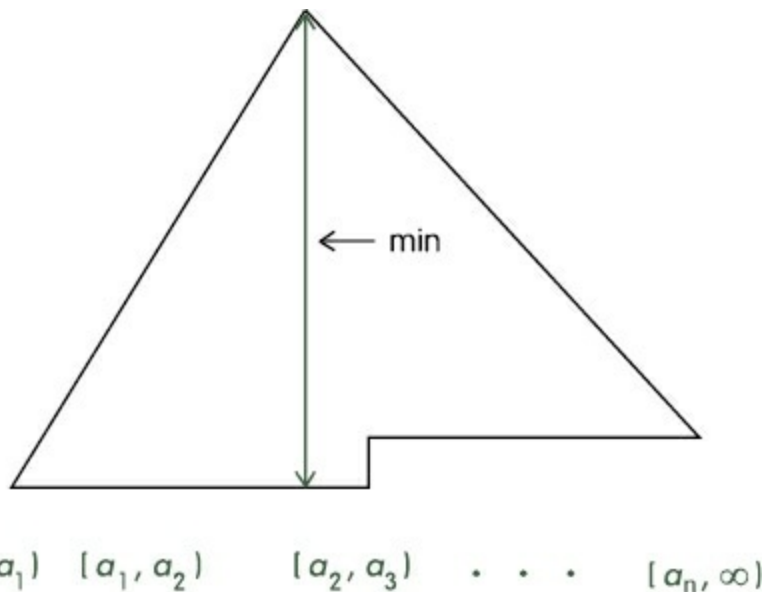
Algorytm binarnego umieszczania wykonuje co najwyżej $\lfloor \log_2 n \rfloor + 1$ porównań na ciąg złożonym z n elementów — dzięki temu, że w każdym kroku przeszukiwany ciąg staje się w przybliżeniu o połowę krótszy. Liczba dzielení przez dwa liczby n , sprowadzających n do wartości 1, wynosi właśnie $\lfloor \log_2 n \rfloor$, a ponieważ niekiedy dążymy do osiągnięcia pustego ciągu — zwiększamy tę liczbę o 1. Naszkicujemy teraz rozumowanie, które dowodzi optymalności tego algorytmu — są to jednak rozważania nieco trudniejsze.

Porównaj tylko: na uporządkowanym ciągu o długości 1000 elementów algorytm binarnego przeszukiwania wykonuje 10 porównań, podczas gdy algorytm liniowego przeszukiwania może wykonać 1000 porównań, czyli 100 razy więcej!

Ćwiczenie 9.11. Posłuż się programem UmieszczanieBinarne do wykonania eksperymentów z algorytmem binarnego umieszczania dla różnych wartości elementu y . Określ liczbę kroków iteracji w każdym przypadku — w tym celu umieść w odpowiednich miejscach programu instrukcje, służące do zliczania iteracji. Przekonaj się, że liczba kroków nie jest większa niż $\lfloor \log_2 n \rfloor + 1$. ■

T

Posłużymy się teraz drzewem algorytmu podobnym do tych, jakie stosujemy w analizie problemu porządkowania (zobacz punkt 4.1 oraz punkt 10.4), by wykazać, że algorytm binarnego umieszczania jest optymalny. W drzewie algorytmu umieszczania każdy wierzchołek pośredni zawiera operację porównania, a liście odpowiadają możliwym wynikom jego działania. Wyników tych jest tyle, ile przedziałów, w których może się znajdować dana wartość elementu y . Zatem gdy przeszukiwany ciąg zawiera n elementów, takich przedziałów jest $n + 1$. Ich końcami są: $-\infty$, n elementów tego ciągu, czyli a_1, a_2, \dots, a_n , oraz $+\infty$ (zobacz rysunek 9.3).



Rysunek 9.3. Schematyczna postać drzewa algorytmu przeszukiwania ciągu a $[1..n]$

Z interpretacji drzew algorytmów wynika, że drzewo najlepszego algorytmu ma najmniejszą możliwą wysokość. Zatem pytanie o to, jaki jest najlepszy algorytm umieszczania, można sformułować następująco: jaka jest najmniejsza wysokość drzewa algorytmu, które ma przynajmniej $n + 1$ liści? Ponieważ poziom k drzewa algorytmu zawiera 2^k wierzchołków, pytamy więc, jaka jest najmniejsza wartość k , aby wszystkie $n + 1$ wierzchołki odpowiadające możliwym odpowiedziom algorytmu zmieściły się na poziomie k i ewentualnie na poziomie $k - 1$, czyli jaka jest najmniejsza wartość k , aby była spełniona nierówność:

$$2^k \geq n + 1.$$

Po zlogarytmowaniu obu stron otrzymujemy nierówność, w której dodatkowo zastępujemy prawą stronę przez powagę z logarytmu, gdyż liczba poziomów ma być liczbą naturalną:

$$k \geq \lceil \log_2(n+1) \rceil \quad (9.2)$$

Z nierówność (9.2) wynika, że jakikolwiek algorytm umieszczania elementu w uporządkowanym ciągu złożonym z n elementów wykonuje przynajmniej

$$\lceil \log_2(n+1) \rceil \text{ porównań. } \blacksquare$$

Ćwiczenie 9.12. Sprawdź na przykładach wybranych wartości liczby n , m.in. dla $n = 5, 6, 7, 8, 9$, że wielkości: $\lfloor \log_2 n \rfloor + 1$ — złożoności binarnego algorytmu umieszczania, oraz $\lceil \log_2(n+1) \rceil$ — dolnego oszacowania złożoności

jakiegokolwiek algorytmu umieszczania, są sobie równe! ■

Stwierdzenie z ostatniego ćwiczenia jest prawdziwe dla każdego n , zatem w algorytmie binarnego umieszczania jest wykonywanych tyle porównań, ile musi wykonać (w najgorszym przypadku danych) jakikolwiek algorytm umieszczania, czyli uzasadniliśmy, że jest to algorytm optymalny.



Polecamy rozdział 10. w książce [Piramidy], poświęcony również przeszukiwaniu zbiorów uporządkowanych.

9.3. Przeszukiwanie interpolacyjne

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

W niektórych encyklopediach i słownikach opisany obok sposób odszukiwania w nich wyrazów jest ułatwiony na pierwszym etapie przez tzw. „indeks na kciuk” — wycięcia po prawej stronie książki z miejscem na kciuk, oznaczone kolejnymi literami alfabetu.

Wyobraźmy sobie, że mamy znaleźć w książce telefonicznej numer telefonu Pana Alfreda Bogusza. 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.

W jaki sposób można uwzględnić powyższą obserwację w algorytmie przeszukiwania? Dalsze rozważania, dla uproszczenia, przeprowadzimy w odniesieniu do przeszukiwania ciągów liczb. 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. Dokładniej, w algorytmach binarnego przeszukiwania następny punkt w przedziale o końcach *lewy* i *prawy* znajdujemy ze wzoru:

$$s = \frac{\textit{lewy} + \textit{prawy}}{2},$$

który można zapisać w postaci:

$$s = \textit{lewy} + \frac{1}{2}(\textit{prawy} - \textit{lewy}) \quad (9.3)$$

Wynika stąd, że w algorytmach binarnych postępujemy tak, jakby poszukiwana liczba znajdowała się w połowie przeszukiwanego przedziału, gdyż do lewego końca przedziału *lewy* dodajemy połowę długości przedziału. Zatem w rzeczywistości nie jest uwzględniana wartość liczby *y*. A na czym miałyby polegać jej uwzględnienie? Powinniśmy sprawdzać w przeszukiwanym ciągu takie miejsce, które jest tym bliżej lewego końca przedziału, im bardziej wartość *y* jest bliższa wartości elementu w lewym końcu przedziału. Zatem we wzorze (9.3) powinniśmy zastąpić proporcję $1/2$ stosunkiem odległości elementu *y* od elementu w lewym końcu przedziału do rozpiętości całego przedziału. Otrzymujemy stąd następującą modyfikację wzoru (9.3), uwzględniającą wartość *y* oraz wartości elementów w przeszukiwanym ciągu:

$$s = \textit{lewy} + \frac{y - a_{\textit{lewy}}}{a_{\textit{prawy}} - a_{\textit{lewy}}}(\textit{prawy} - \textit{lewy}) \quad (9.4)$$

Algorytm, będący modyfikacją algorytmu binarnego umieszczania, w której uwzględniono wzór (9.4), nazywa się **algorytmem interpolacyjnego umieszczania**. Nazwa ta jest uzasadniona tym, że następny element w ciągu jest określany z uwzględnieniem wielkości poszukiwanego elementu w stosunku do rozpiętości wszystkich elementów w ciągu. W tym algorytmie wprowadzamy dodatkowe zmiany związane z innym sposobem obliczania wartości *s*. Po pierwsze musimy zapewnić, że wartość *s* nie jest większa niż prawy koniec przedziału *prawy*. Ponadto kończymy obliczenia, gdy $a_{\textit{lewy}} = y$, gdyż wtedy *s* nie ulega już zmianie w następnym kroku, zobacz wzór (9.4). Z kolei, dla poprawności obliczeń (by nie dzielić przez zerową rozpiętość przedziału), zakładamy, że ciąg przeszukiwany w tym algorytmie nie zawiera takich samych elementów (zobacz wzór (9.4) i określenie *s* w kroku 2.). To ostatnie założenie nie jest zbyt ograniczające — hasła w słownikach i w encyklopediach czy pozycje w książkach telefonicznych na ogół się nie powtarzają — dwaj abonenci o tych samych imionach i nazwiskach mieszkają zwykle pod różnymi adresami.

Algorytm interpolacyjnego umieszczania

Dane: Uporządkowany ciąg liczb w tablicy $a[k..l]$, gdzie $k \leq l$, to znaczy $a_k \leq$

$a_{k+1} \leq \dots \leq a_l$; oraz element y spełniający nierówność $y \geq a_k$.

Wynik: Miejsce dla y w ciągu $a[k..l]$, czyli takie s , że $a_s = y$, jeśli y jest równe jakiemuś elementowi przeszukiwanego ciągu, lub s jest największą liczbą taką, że $a_s \leq y$.

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

Krok 2.

$$s := \min \left\{ lewy + \left\lceil \frac{y - a_{lewy}}{a_{prawy} - a_{lewy}} (prawy - lewy) \right\rceil, prawy \right\}.$$

Jeśli $a_s \leq y$, to $lewy := s$, a w przeciwnym razie $prawy := s - 1$.

Jeśli $lewy = prawy$ lub $a_{lewy} = y$, to zakończ algorytm, w przeciwnym razie powtórz krok 2. ■

Efektywność

Zadanie 9.1. Porównaj efektywność algorytmów binarnego i interpolacyjnego umieszczania. W tym celu wygeneruj ciąg $n = 50$ losowych liczb całkowitych z przedziału $[1, 100]$, uporządkuj je i zastosuj oba algorytmy do umieszczenia w tym ciągu liczb: 10, 30, 49, 70 i 95, a więc liczb znajdujących się w różnych miejscach przedziału przeszukiwania. ■

Wyniki naszych obliczeń są zawarte w tabeli 9.1.

Tabela 9.1. Wyniki porównania efektywności algorytmów binarnego i interpolacyjnego umieszczania dla danych opisanych w zadaniu 9.1. Zamieszczono liczby iteracji w obu algorytmach dla pięciu różnych ciągów

y Binarne umieszczanie Interpolacyjne umieszczanie

10 6 6 6 6 6 2 3 2 2 2

30 6 6 6 6 5 3 2 2 2 2

49 5 6 6 6 6 3 3 3 2 2

70 6 5 5 6 6 3 3 3 2 1

95 6 6 6 6 6 2 2 3 1 3

Dowodzono, że w algorytmie interpolacyjnego umieszczania wykonuje się przeciętnie liczbę porównań równą $\lceil \log_2 \log_2 n \rceil$, gdzie n jest początkową

długością przeszukiwanego ciągu. Gdy $n = 50$, ta liczba ma wartość 3. Zatem jest to algorytm szybszy od binarnego umieszczania, przynajmniej na podstawie porównania funkcji złożoności. W praktyce komputerowe realizacje tych algorytmów nie wykazują tak dużej przewagi interpolacyjnego umieszczania nad binarnym, zwłaszcza dla niewielkich wartości n , z następujących powodów:

- liczba $\lceil \log_2 n \rceil$ — złożoność algorytmu binarnego umieszczania jest już dostatecznie mała, jej logarytm nie jest więc o wiele mniejszy;
- w algorytmie interpolacyjnego umieszczania są wykonywane dodatkowe operacje arytmetyczne, które zwiększają czas jego działania.

Te uwagi o mniejszej niż przewidywana efektywności metody interpolacyjnego umieszczania odnoszą się do jej komputerowej realizacji. Stosowana intuicyjnie przez każdego z nas podczas wyszukiwania — w encyklopediach, książkach, katalogach bibliotecznych i innych zbiorach uporządkowanych alfabetycznie — jest jednak metodą znacznie szybszą niż metody binarne.

9.4. Znajdowanie miejsca zerowego funkcji metodą połowienia przedziału

Binarne metody poszukiwań elementów w uporządkowanym ciągu mają swój odpowiednik wśród metod przeglądania punktów przedziału w poszukiwaniu punktu o specjalnych własnościach. W tym przypadku również korzystamy z uporządkowania elementów, wśród których szukamy — tymi elementami są bowiem wszystkie punkty (a dokładniej, liczby) w przedziale, uporządkowane zgodnie ze swoimi wartościami.

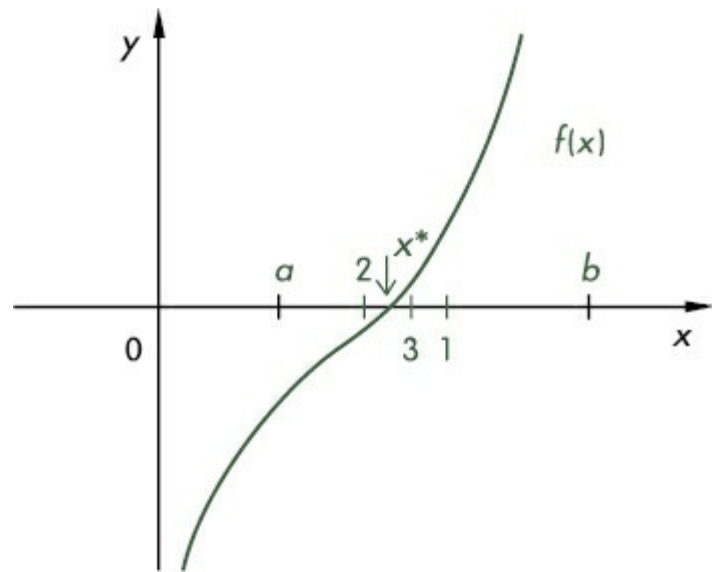
Metodę podziału na pół zbioru, w którym poszukujemy elementu, zastosujemy teraz do znajdowania miejsca zerowego funkcji w zadanym przedziale.

Niech $f(x)$ będzie funkcją ciągłą (czyli wykres tej funkcji jest linią ciągłą) w przedziale domkniętym $[a, b]$ oraz dodatkowo założmy, że spełniony jest warunek $f(a) \cdot f(b) < 0$, czyli na końcach przedziału funkcja f ma przeciwne znaki. Te warunki oznaczają, że w przedziale $[a, b]$ znajduje się punkt x^* spełniający równość $f(x^*) = 0$, czyli x^* jest **miejscem zerowym** funkcji f , ponieważ aby funkcja ciągła mogła mieć na końcach przedziału przeciwne znaki, jej wykres musi w tym przedziale przeciąć oś Ox — ten punkt przecięcia jest właśnie miejscem zerowym funkcji (rysunek 9.4).

Metoda znajdowania x^* takiej funkcji jest bardzo prosta i narzuca się sama:

- podziel przedział punktem znajdującym się w połowie,
- oblicz znak funkcji f w tym punkcie,

► z dwóch podprzedziałów wybierz ten, na którego końcach funkcja f ma nadal przeciwne znaki.



Rysunek 9.4. Wykres funkcji ciągłej z miejscem zerowym w przedziale $[a, b]$. Zielone punkty na osi, oznaczone cyframi, są trzema kolejnymi przybliżeniami miejsca zerowego tej funkcji, znalezionymi metodą połowienia przedziału

Ideę metody połowienia przedziału można odnaleźć w dowodach wielu twierdzeń matematycznych, w których konstruowany jest ciąg zbieżny — końce przedziałów, generowanych w tej metodzie, tworzą właśnie taki ciąg.

Oznaczmy przez $[a_i, b_i]$ ciąg kolejnych przedziałów generowanych w tej metodzie, gdzie $[a_0, b_0] = [a, b]$. Podobnie jak w iteracyjnej metodzie znajdowania wartości pierwiastka kwadratowego (omówionej w punkcie 7.7) musimy określić, kiedy przerwać obliczenia. Do wyboru mamy trzy kryteria, z których dwa pierwsze są podobne jak w tamtej metodzie:

1. Różnica między kolejnymi przybliżeniami wartości zera funkcji jest mniejsza niż przyjęta przez nas dokładność obliczeń Eps , czyli $b_i - a_i < Eps$.
2. Liczba wykonanych iteracji osiągnęła określoną przez nas granicę $MaxIter$.
3. Wartość funkcji w kolejnym przybliżeniu jest dostatecznie bliska zeru, czyli mniejsza niż zadana liczba $Eps1$; ten warunek możemy zapisać w postaci:

$$|f((a_i + b_i)/2)| \leq Eps1.$$

Zauważmy, że dwa pierwsze kryteria są w pewnym sensie zależne od siebie. Jeśli bowiem znamy końce przedziału a i b , to na podstawie liczby wykonanych iteracji, czyli podziałów tego przedziału, możemy określić, jaka jest długość bieżącego przedziału. Po pierwszej iteracji ten przedział jest o połowę krótszy, po drugiej — cztery razy krótszy, po trzeciej jest już jedną ósmą początkowego przedziału itd. Ogólnie po i iteracjach przedział ma długość:

$$\frac{b-a}{2^i}$$

Wiedząc, że ta liczba ma być mniejsza od Eps , można obliczyć, ile trzeba wykonać iteracji, czyli jakie musi być i — wynosi ono przynajmniej $\log_2((b-a)/Eps)$. W związku z tym za kryterium zakończenia poszukiwań miejsca zerowego metodą połowienia przedziału przyjmujemy warunki określone w kryteriach 1. i 3.

Algorytm znajdowania miejsca zerowego funkcji metodą połowienia przedziału

Dane: Funkcja $f(x)$ ciągła w przedziale domkniętym $[a, b]$ i spełniająca nierówność $f(a) \cdot f(b) < 0$. Dodatkowo dane są dwie dokładności obliczeń: Eps i $Eps1$.

Wynik: Przybliżenie x^* miejsca zerowego funkcji f w przedziale $[a, b]$ spełniające warunki: x^* należy do przedziału $[a_i, b_i]$ spełniającego kryterium 1. lub kryterium 3. powyżej.

Krok 1. Przyjmij $a_0 := a$ i $b_0 := b$ oraz oznacz $fa := f(a)$. Niech $i := 0$.

Krok 2. Przyjmij $s := (a_i + b_i)/2$ i $fs := f(s)$.

Krok 3. Jeśli $b_i - a_i < Eps$ lub $|fs| \leq Eps1$, to przyjmij $x^* := s$ i zakończ algorytm.

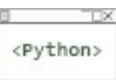
Krok 4. Zwiększ $i := i + 1$. Jeśli $fa \cdot fs < 0$, to przyjmij $a_i := a_{i-1}$ oraz $b_i := s$, a w przeciwnym razie $a_i := s$ oraz $b_i := b_{i-1}$ i $fa := fs$. Wróć do kroku 2. ■

Poniżej zamieszczamy opisy funkcji Zero w językach Pascal i Python. W opisie w języku Python ilustrujemy dodatkową definicję funkcji, której miejsca zerowe mają być wyznaczane.



```
function Zero(a,b:real; Eps,Eps1:real):real;
  var fa,s,fs:real;
begin
  fa := f(a);
  repeat
    s := (a + b)/2; fs := f(s);
    if fa * fs < 0 then b := s
    else begin a := s; fa := fs end
  until (b - a < Eps) or (abs(fs) < Eps1);
```

```
Zero := s  
end; {Zero}
```



```
def f(x):  
    return x*x - 2  
  
def Zero(a,b,Eps,Eps1):  
    fa,fs = f(a), f((a + b)/2);  
    while b-a >= Eps or (abs(fs) >= Eps1):  
        s = (a + b)/2  
        fs = f(s)  
        if fa*fs < 0:  
            b = s  
        else:  
            a,fa = s,fs  
    return s
```

Algorytm połowienia przedziału stanowi dobrą ilustrację zasady dziel i zwyciężaj, jest to jednak wolno zbieżna metoda znajdowania przybliżonej wartości miejsca zerowego funkcji w tym sensie, że dla wybranej dokładności wykonuje sporą liczbę iteracji.



Szybciej zbieżne metody znajdowania miejsc zerowych funkcji są opisane w opracowaniach [EI-I, EI-II, EI-III], a ich działanie może być zilustrowane za pomocą pakietu EI (zobacz również demonstracje zamieszczone w materiałach dołączonych do poradnika [EI-III] oraz na stronie [Oprog]). ■

9.5. Zadania i problemy

Zadanie 9.2. Podaj, w wybranej przez siebie reprezentacji, opis iteracyjnej wersji rekurencyjnego algorytmu znajdowania maksimum i minimum jednocześnie. Jeśli potrafisz programować w języku Pascal/Python, zamień procedurę/funkcję rekurencyjną MinMaxRek na procedurę/funkcję iteracyjną. Zauważ na tym przykładzie, jak zwarty i elegancki jest rekurencyjny zapis algorytmu w porównaniu z opisem iteracyjnym. Tak jest w przypadku większości algorytmów budowanych zgodnie z zasadą dziel i zwyciężaj.

Zadanie 9.3. Posługując się programem w języku Pascal lub Python, znajdź

miejsca zerowe funkcji $f(x) = x^3 - 6x^2 + 11.75x - 7.5$ w przedziale $[1, 3]$. Na końcach tego przedziału ta funkcja ma przeciwne znaki, ale w podanym przedziale ma trzy zera. Jaki zaproponowałbyś sposób szukania miejsc zerowych funkcji, aby w takim przypadku, jak ten, nie pominąć żadnego zera?

Miałeś okazję dowiedzieć się:

- ▶ jaki jest ogólny **schemat metody dziel i zwyciężaj**;
 - ▶ **jak rekurencja pomaga** w prosty sposób zapisać odwoływanie się do rozwiązań, które dopiero konstruujemy;
 - ▶ **na czym polegają metody połowienia** w rozwiązywaniu problemów;
- oraz poznać algorytmy rozwiązywania następujących problemów,
działające na zasadzie dziel i zwyciężaj:
- ▶ **znajdowania maksimum i minimum jednocześnie** — ponownie;
 - ▶ **wyszukiwania i umieszczania** elementów w ciągu uporządkowanym, metodą binarną oraz metodą interpolacyjną;
 - ▶ **znajdowania miejsca zerowego funkcji** metodą połowienia przedziału.

Rozdział 10. Porządkowanie ciągu elementów

Tu poznasz przykłady posłużenia się **zasadą dziel i zwyciężaj** przy tworzeniu najbardziej efektywnych algorytmów porządkowania ciągów metodami:

- ▶ **przez umieszczanie** elementów w ciągu uporządkowanym;
- ▶ **przez scalanie ciągów uporządkowanych**;

oraz:

- ▶ **najszybszą w praktyce metodę porządkowania** ciągów.

Na końcu rozdziału możesz **uporządkować** swoje **wiedomości i umiejętności** związane z... porządkowaniem.

W tym rozdziale posługujemy się zasadą dziel i zwyciężaj przy konstruowaniu algorytmów porządkowania ciągów. W punkcie 10.1 wykorzystujemy znane już nam algorytmy umieszczania, zaś w punkcie 10.2 i 10.3 przedstawiamy dwa nowe algorytmy, które są klasycznymi przykładami użycia tej zasady i posłużenia się rekurencją w jej realizacji. Przy tej okazji omawiamy operację scalania ciągów (punkt 10.2.1), którą można odnaleźć w wielu innych działaniach komputerów. Przedstawione w tym rozdziale algorytmy porządkowania należą do najefektywniejszych.

O najważniejszych aspektach problemu porządkowania piszemy ponownie w punkcie 10.4 — można powiedzieć, że porządkujemy tam wiadomości i umiejętności algorytmiczne wiążące się z porządkowaniem.

Ponownie polecamy aplikację sieciową *Sortowanie* [Oprog], w której można zapoznać się z działaniem podstawowych algorytmów porządkowania, jak i porównać ich działanie na wybranych danych.

10.1. Porządkowanie przez umieszczanie

Algorytmy umieszczania elementu w uporządkowanym ciągu można wykorzystać do... porządkowania ciągu. Ten sposób porządkowania stosujemy dość często w praktycznych sytuacjach, zwłaszcza wtedy, gdy elementy zbioru, który mamy uporządkować, napływają sukcesywnie. Tak jest na przykład w grze w karty, gdy niecierpliwie bierzemy każdą kolejną kartę od rozdającego i umieszczamy w ręce. Jaką stosujemy wtedy na ogół metodę? Staramy się wstawić nową kartę w odpowiednie miejsce, wśród już uporządkowanych kart otrzymanych dotychczas. Taka metoda nazywa się **porządkowaniem przez**

umieszczanie. Operacja umieszczania elementu w ciągu uporządkowanym ma jednak dość poważną wadę. Przypomnij sobie, ile razy, kompletując rękę kart, musiałeś posługiwać się nawet dwiema rękami, by wstawić kolejną kartę — jest to spowodowane koniecznością „rozepchnięcia” dotychczas skompletowanych kart. Niestety, w przypadku realizacji tej metody z ciągiem znajdującym się w tablicy lub w liście, ta trudność nie znika, a wręcz przeciwnie, czas poświęcony na „rozpychanie” ciągu może zdominować czas działania algorytmu porządkującego w ten sposób.

Przedstawiliśmy dwie metody umieszczania elementów w ciągu uporządkowanym: liniową (zobacz punkt 5.1.2 i problem 5.2) oraz binarną (zobacz punkt 9.2). Pierwsza polega na przeglądaniu ciągu od jednego z jego końców, a druga — na przeszukiwaniu ciągu przez jego połowienie. Te metody różnią się efektywnością. Dla ciągu złożonego z n elementów, stosując metodę liniową — w najgorszym przypadku danych wykonujemy n porównań, a stosując metodę binarną — proporcjonalnie do wartości logarytmu z n . Nie ma to jednak wpływu na złożoność rozsuwania ciągu, gdy trzeba umieścić w nim nowy element.

Na rysunku 10.1 ilustrujemy metodę porządkowania przez umieszczanie. Kolejno umieszczane liczby są podane w prawej kolumnie. Wstawiamy je w odpowiednie miejsca, znajdowane w trakcie liniowego lub binarnego przeszukiwania ciągu już uporządkowanych liczb. To miejsce może się znajdować na końcu tego ciągu — wtedy nowy element po prostu dołączamy do ciągu (np. tak zrobiliśmy z elementami 4 i 10), może się znajdować gdzieś w środku ciągu (np. jak z miejscami dla 7 i 9) lub na jego początku (np. dla 2 i 1). W dwóch ostatnich przypadkach część ciągu musimy przesunąć w prawo — złożoność wykonania tego kroku nie zależy jednak od wyboru algorytmu umieszczania.

^3										:2
2	3^									:4
2	3	4^								:10
^2	3	4	10							:1
1	2	3	4^	10						:7
1	2	3	4	7^	10					:9
1	2	3	4	7	9	10				

Rysunek 10.1. Przykład porządkowania przez umieszczanie. Strzałki wskazują miejsce wstawienia do ciągu elementu znajdującego się w prawej kolumnie

Ćwiczenie 10.1. Zastosuj naszkicowaną metodę porządkowania przez umieszczanie do ustawienia ciągu liter KSIAŻKAOALGORYTMACH w porządku

alfabetycznym. (Zobacz uwagę zapisaną pod tekstem ćwiczenia 6.2 w punkcie 6.1). ■

W przedstawionym opisie algorytmu porządkowania przez umieszczanie nie precyzujemy, jaką metodą jest znajdowane miejsce dla nowego elementu.

Algorytm porządkowania przez umieszczanie

Dane: Liczba naturalna n i ciąg n liczb x_1, x_2, \dots, x_n .

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Krok 1. Dla $i = 2, 3, \dots, n$ wykonaj kroki 2. i 3.

Krok 2. Jeśli $x_i < x_1$, to przyjmij $k := 0$, w przeciwnym razie, wśród liczb $1, 2, \dots, i - 1$, znajdź największe takie k , że $x_k \leq x_i$.

Krok 3. Elementy x_{k+1}, \dots, x_{i-1} przesun w prawo na pozycje x_{k+2}, \dots, x_i . Element x_i umieść na zwolnionej pozycji $k+1$. {Uważaj na kolejność przenoszenia elementów — zacznij od prawej.} ■

Ćwiczenie 10.2. Dla dowolnej liczby naturalnej n podaj przykład ciągu złożonego z n liczb, dla którego algorytm porządkowania przez umieszczanie wykonuje $n(n-1)/2$ przestawień elementów. ■

Przypomnijmy: **wartownik** jest elementem ciągu, którego rola polega na „pilnowaniu” końca ciągu, a dokładniej — na powstrzymywaniu przeszukiwania przed wyjściem poza ciąg.

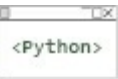
W kroku 2. algorytmu można zastosować albo liniowy algorytm umieszczania, albo binarny. Ten krok trzeba jednak wykonać bardzo uważnie, aby w poszukiwaniu miejsca dla nowego elementu uwzględnić sytuację, gdy ten element ma być umieszczony przed pierwszym z już uporządkowanych elementów. Odnosi się to do obu algorytmów umieszczania — zobacz np. opis algorytmu binarnego umieszczania (punkt 9.2), w którym zakłada się, iż element, dla którego szukamy miejsca, nie może być mniejszy niż x_1 , tj. pierwszy element ciągu. Istnieje jeden wspólny dobry sposób spełnienia tego założenia — użyć **wartownika** (punkt 5.1.1). W tym przypadku ustawiamy go na pozycji dodatkowego elementu x_0 i wystarczy, gdy jego wartość będzie mniejsza od wartości wszystkich elementów porządkowanego ciągu. Jeśli porządkowane liczby są dodatnie, to wartownikiem może być liczba ujemna. Wtedy, jeśli wstawiany element ciągu (który jest dodatni) będzie mniejszy od x_1 , to wartownik będzie wstrzymywał zarówno liniowe, jak i binarne umieszczanie przed wyjściem poza lewy koniec ciągu.

Poniżej podajemy opis algorytmu porządkowania przez umieszczanie w obu językach. Zakłada się, że porządkowane są ciągi nieujemnych liczb, a zatem wartownikiem może być liczba -1 — jest on umieszczony w elemencie o

indeksie 0.



```
procedure InsertionSort(n:integer; var x:Tablica0n);  
  var i,j,k,y:integer;  
begin  
  x[0] := -1; {Wartownik przed pierwszym elementem.}  
  for i := 2 to n do begin  
    y := x[i];  
    k := i - 1;  
    while y < x[k] do k := k - 1;  
    for j := i - 1 downto k + 1 do x[j+1] := x[j];  
    x[k+1] := y  
  end {for}  
end; {InsertionSort}
```



```
def InsertionSort(x):  
  n = len(x)  
  x[0] = -1  
  for i in range(2,n):  
    y = x[i]  
    k = i - 1  
    while y < x[k]:  
      k = k - 1  
    for j in range(i-1,k,-1):  
      x[j+1] = x[j]  
    x[k+1] = y
```

Zadanie 10.1. Zmodyfikuj wybraną realizację porządkowania przez umieszczanie tak, aby miejsce dla kolejnego elementu w uporządkowanym ciągu było znajdowane metodą binarnego umieszczania — patrz punkt 9.2. ■

Ćwiczenie 10.3. W wybranym przez siebie języku, uzupełnij podany przez nas program oraz program utworzony przez siebie w zadaniu 10.1 o zliczanie porównań i zamian elementów. Następnie porównaj liczby tych operacji w dwóch realizacjach algorytmu porządkowania przez umieszczanie — w jednej z liniową, a w drugiej — z binarną metodą umieszczania, na ciągach o różnych

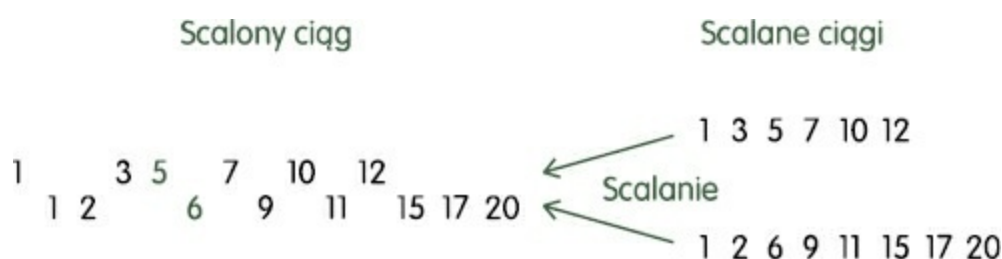
długościach. ■

10.2. Porządkowanie przez scalanie

Jedną z podstawowych operacji, wykonywanych na zbiorach informacji, jest ich scalanie. Termin „scalanie” jest jednak używany w wielu różnych znaczeniach. W najprostszym sensie, scalić dwa zbiory informacji to znaczy dopisać drugi zbiór do końca pierwszego. Dalej będziemy zakładać, że scalane zbiory są uporządkowanymi ciągami liczb i wynik scalania ma być również ciągiem uporządkowanym.

10.2.1. Scalanie ciągów uporządkowanych

Scalenie dwóch uporządkowanych ciągów w jeden ciąg uporządkowany może być wykonane w bardzo prosty sposób: patrzymy na początki danych ciągów i do tworzonego ciągu przenosimy mniejszy z elementów czołowych lub którykolwiek z nich, gdy są równe. Ilustrujemy to przykładem na rysunku 10.2.



Rysunek 10.2. Przykład scalania dwóch ciągów uporządkowanych

Kolejność przenoszenia elementów z ciągów zależy od ich wartości, które mogą powodować, że jeden ciąg będzie znacznie wcześniej przeniesiony niż drugi.

Ćwiczenie 10.4. Podaj odpowiednie przykłady dwóch uporządkowanych ciągów, w trakcie scalania których jeden ciąg jest znacznie szybciej w całości przeniesiony do tworzonego ciągu niż drugi. ■

Czy można określić, ile porównań należy wykonać, aby scalić dwa ciągi? Z ostatniego ćwiczenia wynika, że liczba porównań może zależeć od wartości elementów. Zastanówmy się więc, ile najwięcej porównań musimy wykonać, by scalić dwa ciągi? Rozważanie różnych możliwych układów wartości elementów w obu scalanych ciągach nie daje szybkiej odpowiedzi na to pytanie. Spójrzmy natomiast od strony tworzonego ciągu. Z wyjątkiem elementu przeniesionego do niego na końcu, przeniesienie każdego innego elementu może być związane z wykonaniem porównania między elementami danych ciągów. Tak jest wtedy, gdy w kroku drugim do końca oba scalane ciągi zawierają jeszcze po jednym elemencie. Przypuśćmy więc, że na początku jeden ciąg zawiera k elementów, a drugi — l elementów, razem n , czyli $n = k + l$. Ta dyskusja prowadzi do

konkluzji, że bez względu na licznosci danych ciągów, ich scalenie może wymagać wykonania $n - 1$ porównań i każde z tych porównań jest związane z przeniesieniem jednego elementu, z wyjątkiem elementu, który trafia do scalonego ciągu na końcu.

Z ćwiczenia 10.4 i z tej dyskusji wynika jeszcze jeden wniosek — ponieważ nie potrafimy przewidzieć, który z ciągów i kiedy wyczerpie się w trakcie scalania, tworzony ciąg nie może być umieszczany na miejscu ciągów danych do scalenia, musi więc być tworzony na nowym miejscu. Zatem potrzebna jest dodatkowa pamięć — mówimy wówczas, że ta metoda nie działa „w miejscu” (zobacz punkt 10.4). Ta dodatkowo wykorzystywana pamięć jest słabą stroną operacji scalania.

Algorytm scalania dwóch ciągów uporządkowanych

Dane: Dwa uporządkowane ciągi x i y .

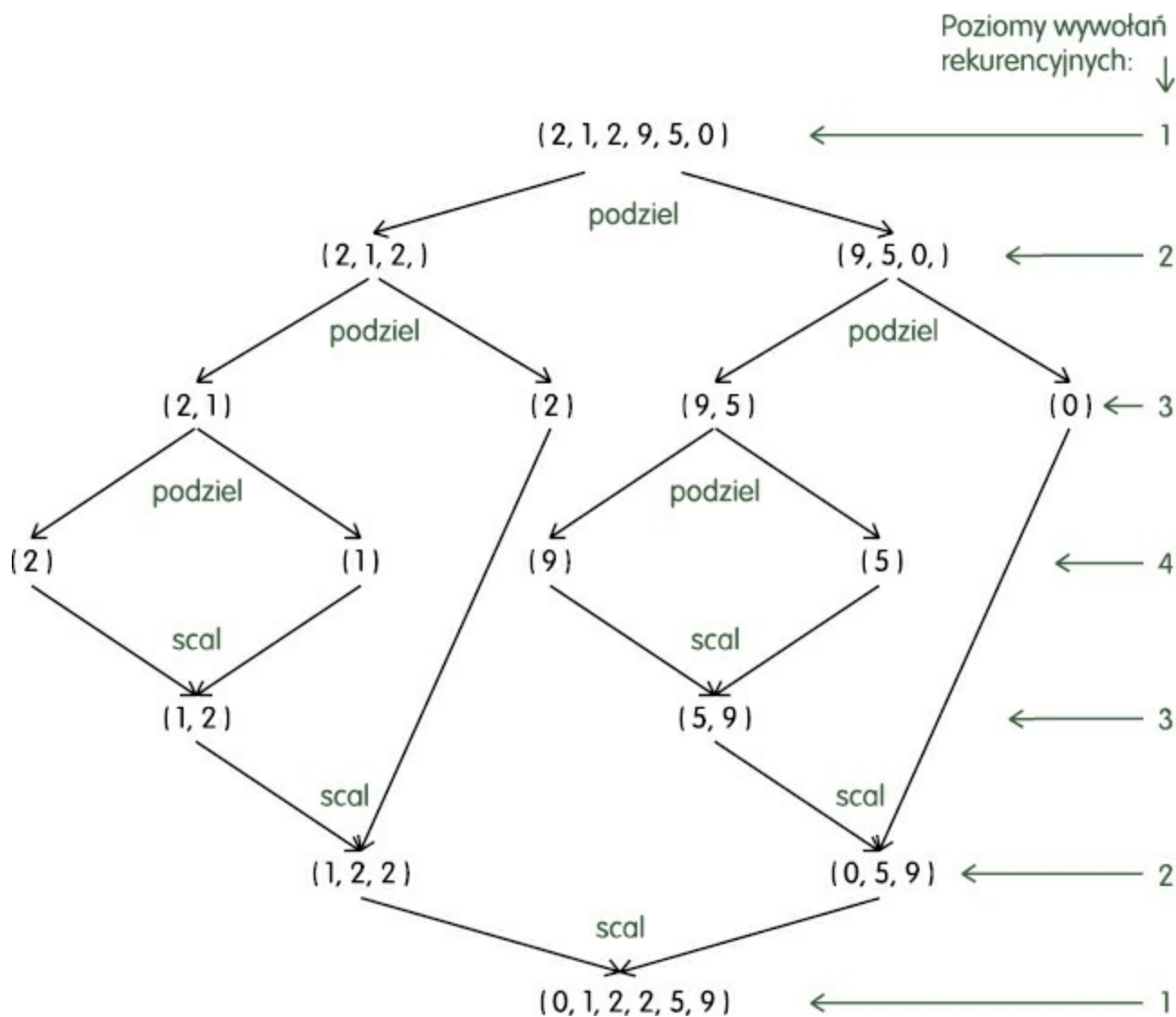
Wynik: Uporządkowany ciąg z , będący scaleniem ciągów x i y .

Krok 1. Dopóki oba ciągi x i y nie są puste, wykonuj następującą operację: przenieś mniejszy z najmniejszych elementów z ciągów x i y do ciągu z .

Krok 2. Do końca ciągu z dopisz elementy pozostałe w jednym z ciągów x lub y . ■

10.2.2. Porządkowanie przez scalanie

Algorytm scalania dwóch uporządkowanych ciągów można zastosować do tworzenia uporządkowanych ciągów z podciągów już uporządkowanych. A czy może być z niego jakiś pożytek przy porządkowaniu dowolnych ciągów? Tak — jeśli potrafimy najpierw uporządkować te podciągi. A czy moglibyśmy założyć, że te podciągi zostały uporządkowane... tą samą metodą? Możemy — i tutaj przydaje się rekurencja. Dodatkowo założymy, że na każdym etapie podciągi mają prawie taką samą długość. Dochodzimy w ten sposób do metody porządkowania ciągu, która, na zasadzie dziel i zwyciężaj, scala dwa prawie równoliczne podciągi, uporządkowane tą samą metodą. Jej działanie można prześledzić na rysunku 10.3. Zauważmy olbrzymie podobieństwo w schematach działania tej metody i rekurencyjnego znajdowania maksimum i minimum jednocześnie — zobacz rysunek 9.1.



Rysunek 10.3. Przykład działania algorytmu porządkowania przez scalanie

Zapiszemy teraz algorytm porządkowania przez scalanie w postaci listy kroków. Z powyższego szkicu oraz z ilustracji na rysunku 10.3 wynika, że porządkowany ciąg jest dzielony w każdym kroku na dwa niemal równej długości podciągi, które rekurencyjnie są porządkowane tą samą metodą. Warunkiem zakończenia rekurencji w tym algorytmie jest sytuacja, gdy ciąg ma jeden element, wtedy nie można go już podzielić na podciągi, nie ma nawet po co — jest to bowiem ciąg już uporządkowany. Zatem powrót z wywołań rekurencyjnych rozpoczyna się z ciągami złożonymi z pojedynczych elementów, które są scalane w ciągi o długości dwa, następnie w ciągi o długości trzy lub cztery elementy itd. Jak zwykle w opisie algorytmu rekurencyjnego, po nazwie algorytmu występuje układ parametrów określających rozwiązywany problem, który jest wykorzystany w treści algorytmu, w odwołaniu do niego samego przy rozwiązywaniu mniejszych podproblemów.

Algorytm porządkowania przez scalanie (l, p, x)

Dane: Ciąg liczb x_l, x_{l+1}, \dots, x_p .

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Krok 1. Jeśli $l < p$, to przyjmij $s := (l + p) \text{ div } 2$ i wykonaj trzy następne kroki.

Krok 2. Zastosuj ten sam algorytm dla (l, s, x) .

Krok 3. Zastosuj ten sam algorytm dla $(s + 1, p, x)$.

Krok 4. Zastosuj algorytm scalania do ciągów x_l, \dots, x_s oraz x_{s+1}, \dots, x_p i wynik umieść z powrotem w ciągu x_l, \dots, x_p . ■

Ćwiczenie 10.5. Użyj algorytmu porządkowania przez scalanie do ustawienia ciągu liter KSIĄŻKAOALGORYTMACH w porządku alfabetycznym. (Zobacz uwagę zapisaną pod tekstem ćwiczenia 6.2 w punkcie 6.1). ■

Jak już wspomnieliśmy przy okazji scalania ciągów — wykonanie tej operacji wymaga dodatkowego miejsca. Można jednak nie zapełniać go w całości, oszczędzając przy tym na liczbie wykonywanych operacji — o tym jest następne zadanie.

Zadanie 10.2. Powtórz poprzednie ćwiczenie z następującą modyfikacją kroku 4., w którym są scalane dwa uporządkowane podciągi. W algorytmie scalania dwóch uporządkowanych ciągów z punktu 10.2.1, w kroku 2., końcowa część jednego z ciągów jest dopisywana do ciągu z . Jeśli są scalane dwa podciągi tego samego ciągu (jak w algorytmie porządkowania przez scalanie), to nie zawsze jest to konieczne. Gdy należy przenieść końcową część pierwszego podciągu, można od razu umieścić ją na końcu tworzonego ciągu (uwaga, trzeba to robić od końca przenoszonych podciągu). Natomiast gdy należy przenieść końcową część drugiego podciągu, można zostawić ją tam, gdzie jest. ■

Modyfikacja kroku 4. opisana w zadaniu 10.2 została uwzględniona w realizacjach algorytmu porządkowania przez scalanie w języku Pascal. W realizacji w języku Python wykorzystaliśmy łatwe dzielenie list na podlisty.



```
procedure MergeSort(Dol,Gora:integer; var x:TablicaIn);  
  var s:integer;  
  procedure Scal(l,s,p:integer);  
    var i,j,k,m:integer;  
        z :TablicaIn;  
  begin  
    i := l; j := s; m := l;  
    while (i < s) and (j <= p) do begin  
      if x[i] <= x[j] then begin  
        z[m] := x[i]; i := i + 1 end  
      else begin z[m] := x[j]; j := j + 1 end;
```

```

    m := m + 1
end; {while}
if i < s then begin
    j := s - 1; k := p;
    while j >= i do begin
        x[k] := x[j];
        k := k - 1; j := j - 1
    end
end;
for i := l to m - 1 do x[i] := z[i]
end; {Scal}
begin {MergeSort}
    if Dol < Gora then begin
        s := (Dol + Gora) div 2;
        MergeSort(Dol,s,x);
        MergeSort(s+1,Gora,x);
        Scal(Dol,s+1,Gora)
    end
end; {MergeSort}

```



```

def Scal(l1,l2,l3):
    i1,i2,i3 = 0,0,0
    n1,n2 = len(l1),len(l2)
    while i1 < n1 and i2 < n2:
        if l1[i1] < l2[i2]:
            l3[i3],i1 = l1[i1],i1+1
        else:
            l3[i3],i2 = l2[i2],i2+1
        i3 = i3 + 1
    while i1 < n1:
        l3[i3],i1,i3 = l1[i1],i1+1,i3+1
    while i2 < n2:
        l3[i3],i2,i3 = l2[i2],i2+1,i3+1
def MergeSort(x):
    n = len(x)
    if n > 1:

```



```

m = n // 2
x1,x2 = x[:m],x[m:]
MergeSort(x1)
MergeSort(x2)
Scal(x1,x2,x)

```

Złożoność

T

Liczbę porównań w algorytmie porządkowania przez scalanie można wyznaczyć, korzystając z jego postaci, podobnie jak w rekurencyjnym algorytmie MaxMinRek (zobacz punkt 9.1). Oznaczmy tę liczbę przez $r(n)$ i założmy (podobnie jak tam), że n jest potęgą liczby 2, czyli $n = 2^k$ dla pewnego k . Wtedy otrzymujemy następującą zależność rekurencyjną:

$$r(n) = \begin{cases} 0, & n=1 \\ 1, & n=2 \\ 2r(n/2) + (n-1) & n>2 \end{cases} \quad (10.1)$$

Gdy $n = 1$, nie wykonujemy żadnego porównania, gdy $n = 2$ — wykonujemy jedno porównanie w kroku 4. podczas scalania ciągów jednoelementowych, a gdy $n > 2$ — wykonujemy dwa razy porządkowanie tą samą metodą ciągów o połowę krótszych (w krokach 2. i 3.) oraz scalamy te ciągi, wykonując co najwyżej $n - 1$ porównań (o jedno mniej niż jest elementów w obu podciągach).

Zależność (10.1) rozwiązujemy podobnie jak zależność (9.1), stosując metodę wstawiania; nie będziemy jednak podawać tutaj całego rozumowania, a jedynie końcową postać funkcji $r(n)$; (zobacz problem 10.1):

$$r(n) = n \log_2 n - n + 1 \quad (10.2)$$

Przytoczyliśmy tutaj postać funkcji złożoności algorytmu porządkowania przez scalanie, gdyż wyróżnia ona ten algorytm wśród innych metod. Dokładniej piszemy o tym w punkcie 10.4.

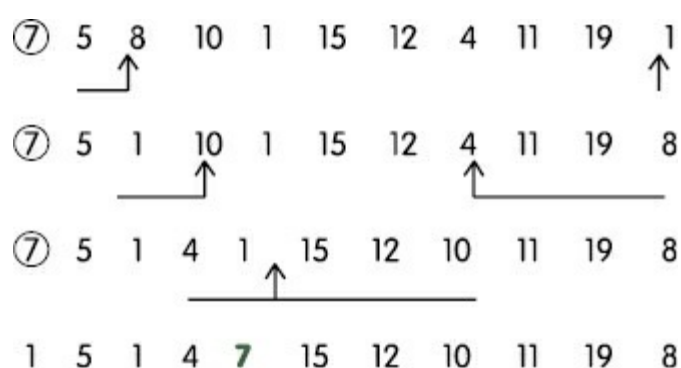
10.3. Szybki algorytm porządkowania

Podajemy tutaj jeszcze jeden algorytm porządkowania, który działa na zasadzie dziel i zwyciężaj. Uchodzi on ponadto za najszybszą w praktyce metodę porządkowania — stąd jego popularna nazwa, **quicksort** — **szybki algorytm porządkowania**. Ten algorytm jest zrealizowany w licznych bibliotekach

programów standardowych i dostępny w wielu komputerach.

Podstawowy krok algorytmu szybkiego polega na podziale porządkowanego ciągu wybranym elementem v na dwie takie części, że w jednej znajdują się liczby nie większe niż v , a w drugiej — liczby nie mniejsze niż v . Po wykonaniu tego kroku element v możemy umieścić między tymi podciągami, gdyż jest to właściwe dla niego miejsce w szukanym uporządkowanym ciągu, a następnie możemy przejść do porządkowania obu podciągów... tą samą metodą.

Musimy teraz odpowiedzieć na dwa pytania: jaki element ciągu wybrać za v oraz jak podzielić tym elementem ciąg na dwa podciągi — najlepiej, gdybyśmy mogli dokonać tego podziału w tym samym miejscu, w którym znajduje się porządkowany ciąg. Często przyjmuje się, że v jest pierwszym elementem dzielonego ciągu i tak przyjmujemy w naszych rozważaniach, przynajmniej na początku. Zilustrujemy teraz na przykładzie pokazanym na rysunku 10.4, w jaki sposób dzieli się ciąg wybranym elementem — jest on otoczony kółkiem.

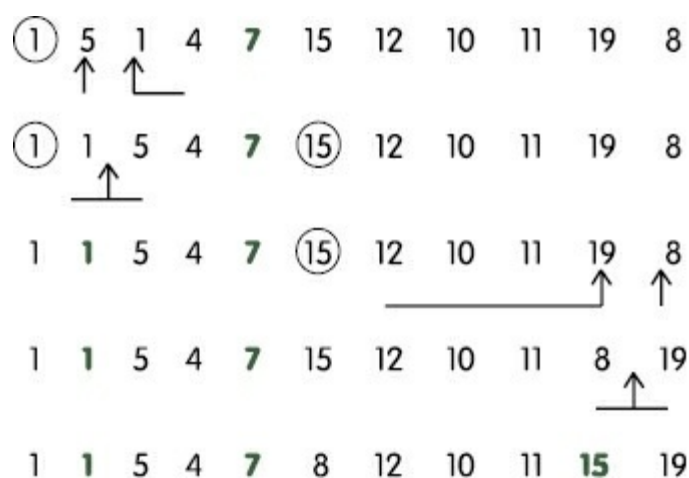


Rysunek 10.4. Przykład działania kroku podziału ciągu w szybkim algorytmie porządkowania

Podział ciągu rozpoczynamy od obu jego końców, posuwając się w kierunku środka ciągu. Ponieważ po lewej stronie chcemy mieć elementy nie większe niż v , w ruchu od lewej strony zatrzymujemy się na elemencie większym od v . Z tego samego powodu, w ruchu od prawej strony zatrzymujemy się na elemencie mniejszym od v . Na rysunku 10.4 te elementy są wskazane strzałkami. Pierwszą parą takich elementów jest: 8 — z lewej strony i 1 — z prawej. Jeśli zamienimy te elementy miejscami, to znajdą się one na odpowiednich miejscach względem elementu v i będziemy mogli pójść dalej, ku środkowi ciągu. W następnym kroku natykamy się na parę: 10 — z lewej strony i 4 — z prawej. Zamieniamy te elementy miejscami i podążamy dalej. Przed znalezieniem następnej pary droga z lewej strony spotyka się z drogą z prawej. W tym momencie możemy przerwać przestawianie, gdyż na lewo od punktu spotkania elementy są nie większe niż v , a na prawo — nie mniejsze niż v . Wykonujemy teraz zamianę v z najbardziej na prawo ustawionym elementem z lewego ciągu — w naszym przykładzie zamieniamy 7 z 1. W ten sposób $v = 7$ zajmie miejsce, które ma w uporządkowanym ciągu, gdyż na lewo są elementy nie większe od niego, a na prawo — nie mniejsze. Na rysunkach elementy znajdujące się już na swoich

miejscach są pogrubione i oznaczone zielonym kolorem.

Dalsze kroki algorytmu szybkiego porządkowania polegają na rekurencyjnym zastosowaniu tej samej metody do obu podciągów, znajdujących się na lewo i na prawo od elementu v — ilustrujemy to na rysunku 10.5, najpierw na lewym, a później na prawym podciągu. Elementami podziału są pierwsze elementy tych podciągów — otoczone kółkiem, a po zakończeniu podziału — oznaczone kolorem zielonym.



Rysunek 10.5. Przebieg wykonania algorytmu podziału na podciągach wyznaczonych w pierwszej iteracji szybkiego algorytmu porządkowania (kontynuacja przykładu z rysunku 10.4)

W następnych krokach tego algorytmu procedura podziału jest stosowana do czterech podciągów znajdujących się w ostatnim wierszu na rysunku 10.5 i oznaczonych kolorem czarnym. Ciągi jednoelementowe, pierwszy i czwarty, nie są dalej dzielone — jest to warunek zakończenia rekurencji. W ciągu (5, 4), podział elementem 5 oznacza zamianę miejscami tych elementów i ustalenie pozycji elementu 5. Następnie jest ustalana pozycja elementu 4 z tego podciągu. W ciągu (8, 12, 10, 11) podział elementem 8 nie zmienia porządku — pozostaje do uporządkowania podciąg (12, 10, 11). W tym celu element 12 jest zamieniany z elementem 11 i na końcu jeszcze 11 z 10. Na rysunku 10.6 są przedstawione w wierszach wyniki kolejnych wywołań rekurencyjnych — każdy następny wiersz zawiera o jeden więcej element oznaczony na zielono; jest to ten element, który w kolejnej iteracji został wybrany do podziału i zajął swoją stałą pozycję w porządkowanym ciągu.

1	1	5	4	7	8	12	10	11	15	19
1	1	5	4	7	8	12	10	11	15	19
1	1	4	5	7	8	12	10	11	15	19
1	1	4	5	7	8	12	10	11	15	19
1	1	4	5	7	8	12	10	11	15	19
1	1	4	5	7	8	11	10	12	15	19
1	1	4	5	7	8	10	11	12	15	19
1	1	4	5	7	8	10	11	12	15	19
1	1	4	5	7	8	10	11	12	15	19

Rysunek 10.6. Kolejno ustalane pozycje elementów w ciągu z rysunku 10.4 i 10.5

Ćwiczenie 10.6. Sprawdź, czy rozumiesz powyższą metodę, ustawiając za jej pomocą ciąg liter KSIĄŻKAOALGORYTMACH w porządku alfabetycznym. (Zobacz uwagę zapisaną pod tekstem ćwiczenia 6.2 w punkcie 6.1). ■

Opiszemy teraz w postaci listy kroków algorytm szybkiego porządkowania. W pierwszym kroku: ciąg jest dzielony na dwie części podaną wyżej metodą, a w dwóch następnych krokach: otrzymane z podziału podciągi są porządkowane tym samym algorytmem. Algorytm jest rekurencyjny, więc w jego nazwie występują parametry określające rozwiązywany problem. Warunek zakończenia rekurencji zależy od długości ciągu — jeśli wynosi ona 1 (czyli, gdy $l = p$), to tego ciągu już dalej nie dzielimy.

Algorytm szybkiego porządkowania (l, p, x)

Dane: Ciąg liczb x_l, x_{l+1}, \dots, x_p .

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Krok 1. Jeśli $l < p$, to przyjmij za element podziału $v := x_l$, i podziel tym elementem dany ciąg. Oznacza to, że v znajdzie się na pozycji elementu x_k , dla pewnego k spełniającego $l \leq k \leq p$, i elementy na lewo będą od niego nie większe, a na prawo — nie mniejsze.

Wykonaj dwa następne kroki.

Krok 2. Zastosuj ten sam algorytm do ($l, k - 1, x$).

Krok 3. Zastosuj ten sam algorytm do ($k + 1, p, x$).

Ustaliliśmy na początku, że elementem służącym w tym algorytmie do podziału porządkowanego ciągu na dwa podciągi jest pierwszy element tego ciągu. Nie zawsze jest to dobry wybór. Jeśli na przykład pierwszy element jest najmniejszym (lub największym) elementem tego ciągu, to krok 1. kończy się podziałem ciągu na dwa podciągi, z których jeden jest złożony jedynie z tego elementu. Biorąc pod uwagę ogólne własności metody dziel i zwyciężaj — a taką metodą jest algorytm szybkiego porządkowania — byłoby dobrze, gdyby element wybierany do podziału zapewniał podział ciągu na prawie równe części. Takim elementem jest mediana ciągu, o której znajdowaniu piszemy w punkcie 5.6. Niestety, nie jest znany szybki algorytm znajdowania mediany i w praktycznych realizacjach szybkiego algorytmu porządkowania nie korzysta się z mediany. W realizacjach, które przedstawiamy poniżej, podział jest wykonywany elementem, który leży w połowie ciągu. Nie będziemy jednak szczegółowo uzasadniać tego wyboru — ma on pewne zalety, o których piszemy w problemie 10.2 (porównaj również problem 10.3). Można przyjąć, że jest to równie odpowiedni element, jak pierwszy, którym posługiwaliśmy się powyżej. Zauważmy, że w opisie algorytmu szybkiego porządkowania nigdzie nie korzysta się z tego, że jest to pierwszy element ciągu.

Ćwiczenie 10.7. Zastosuj szybką metodę porządkowania z elementem środkowym jako elementem dzielącym do ustawienia ciągu liter KSIĄŻKAOALGORYTMACH w porządku alfabetycznym. Porównaj liczbę wykonanych porównań w tym przypadku z liczbą porównań wykonanych w ćwiczeniu 10.6. (Zobacz uwagę zapisaną pod tekstem ćwiczenia 6.2 w punkcie 6.1). ■

Opisany wyżej algorytm szybkiego porządkowania jest zrealizowany w procedurze (funkcji) z dwoma parametrami — dolną i górną granicą indeksów porządkowanego ciągu. Jak wspomnieliśmy, w tej realizacji elementem dzielącym ciąg na dwa podciągi jest element środkowy ciągu (zobacz problem 10.2).



```
procedure QuickSort(Dol,Gora:integer; var x:TablicaIn);
  procedure Sort(d,g:integer);
    var l,p,v,z:integer;
  begin
    l := d; p := g;
    v := x[(d + g) div 2];
    repeat
      while x[l] < v do l := l + 1;
      while v < x[p] do p := p - 1;
```

```

if l <= p then begin
    z := x[l]; x[l] := x[p]; x[p] := z;
    l := l + 1; p := p - 1
end
until l > p;
if d < p then Sort(d,p);
if l < g then Sort(l,g)
end; {Sort}
begin {QuickSort}
    Sort(Dol,Gora)
end; {QuickSort}

```



```

def Sort(d,g):
    l,p = d,g
    v = x[(d+g) // 2]
    while l <= p:
        while x[l] < v:
            l = l + 1
        while v < x[p]:
            p = p - 1
        if l <= p:
            x[l],x[p] = x[p],x[l]
            l,p = l+1,p-1
    if d < p:
        Sort(d,p)
    if l < g:
        Sort(l,g)
def QuickSort(x):
    n = len(x)
    Sort(0,n-1)

```

10.4. Własności algorytmów porządkowania

Problem porządkowania i algorytmy rozwiązujące go tworzą klasyczny dział algorytmiki i o ich ważności nie trzeba chyba nikogo przekonywać. Przypomnijmy tylko, że komputery z jednej strony znaczną część czasu pracy

„spędzają” na porządkowaniu informacji, jakie przechowują, a z drugiej — łatwiej i szybciej korzystają z różnych informacji, gdy są one uporządkowane. Do porządkowania informacji stosuje się algorytmy porządkujące, a przykładami działań, w których uporządkowanie elementów odgrywa istotną rolę dla efektywności ich wykonywania, są wszelkiego rodzaju problemy wyszukiwania i umieszczania elementów w zbiorach danych.

W poprzednich rozdziałach omawialiśmy problem porządkowania, by na przykładach różnych metod jego rozwiązywania zilustrować różne sposoby konstruowania algorytmów, znajdujące zastosowania również przy rozwiązywaniu wielu innych problemów. Opracowaliśmy więc kilka algorytmów porządkowania, a teraz dokładniej omówimy ich podobieństwa i różnice. Może to być przykładem rozważań, gdy dla wybranego problemu dysponujemy kilkoma algorytmami jego rozwiązywania i chcemy poznać ich własności, by wiedzieć, który z nich zastosować dla konkretnych danych.

Przedstawiliśmy sześć algorytmów porządkowania ciągów liczb oraz dwa algorytmy porządkowania elementów, złożonych również z innych znaków niż cyfry. Przyjrzyjmy się teraz nieco bliżej własnościom tych algorytmów i zastosowanym w nich technikom algorytmicznym.

Proponujemy krótki ich przegląd w aplikacji sieciowej *Sortowanie* [Oprog].

Techniki porządkowania

Techniki, zastosowane przez nas do porządkowania elementów, można podzielić na cztery grupy:

- 1.** Metody wynikające wprost z definicji problemu, w których działania w możliwie najprostszy, chociaż nie zawsze najszybszy sposób, prowadzą do spełnienia warunku nałożonego na wynik, a mianowicie elementy mają być ustawione w kolejności rosnącej, od najmniejszego. Przykładami są: algorytm bąbelkowy (punkt 6.1) i algorytm porządkowania przez wybór (punkt 6.2). Podstawowa operacja w pierwszym z tych algorytmów ma charakter lokalny — „naprawia” jedną złą relację między sąsiednimi elementami, a w drugim — polega na znalezieniu kolejnego elementu zgodnego z założonym porządkiem, odnosi się więc do całego ciągu.
- 2.** Metody wykorzystujące drzewa. Algorytm porządkowania na drzewie (punkt 5.4 i problem 5.3) jest metodą, w której drzewo służy do pamiętania wyników porównań oraz do planowania kolejnych kroków algorytmu. Z kolei drzewo algorytmu jest stosowane do przedstawienia niektórych algorytmów porządkujących (rozdział 4.), a także do analizy złożoności obliczeniowej algorytmów oraz problemu porządkowania (zobacz koniec tego punktu).
- 3.** Najbardziej efektywne algorytmy porządkowania przedstawione w tej książce to: algorytm przez umieszczanie (punkt 10.1), przez scalanie (punkt 10.2) oraz algorytm szybki (punkt 10.3). Działają one na zasadzie dziel i

zwycięzaj, która i w tym przypadku „zwycięza” inne techniki.

4. Algorytmy: kubełkowy i porządkowania pozycyjnego (punkt 6.3) wykorzystują wartości porządkowanych elementów i działają bardzo efektywnie, gdy elementy nie przyjmują zbyt wielu różnych wartości. Te algorytmy można stosować do porządkowania obiektów o strukturze bardziej ogólnej niż liczby, np.: słowa (w dowolnym alfabecie), daty, rekordy.

Te techniki nie wyczerpują wszystkich możliwych podejść do porządkowania elementów, zwłaszcza gdy elementy mają jakieś szczególne wartości lub własności. O niektórych innych sposobach porządkowania piszemy w problemach 6.2, 13.25 i 13.32. Pełny przegląd metod porządkowania (sortowania) znajduje się w [Knuth-3].

Porządkowanie stabilne

Przypuśćmy, że została sporządzona lista wszystkich uczniów w szkole zgodnie z alfabetycznym porządkiem ich nazwisk. Taką listę można wykorzystywać w różnych celach. Na przykład na jej podstawie można opracowywać inne, pochodne listy uczniów, na których porządek będzie ustalony zgodnie z innymi kryteriami, takimi jak: oceny — do klasyfikacji postępów w nauce, wyniki sportowe — w rankingu najlepszych sportowców szkoły, ilość zebranej makulatury — do oceny aktywności pozalekcyjnej. Naturalnym oczekiwaniem jest, że jeśli dwóch uczniów ma takie same oceny, wyniki sportowe lub oddało tyle samo makulatury, to występują oni na liście pochodnej, w kolejności alfabetycznej.

Warunek wyrażony w ostatnim zdaniu definiuje pewien rodzaj algorytmów porządkujących, a mianowicie jeśli dwa elementy w porządkowanym ciągu mają taką samą wartość cechy, według której odbywa się porządkowanie, to ich kolejność względem siebie w uporządkowanym ciągu jest taka sama, jak w ciągu danym do uporządkowania. Taki algorytm nazywa się **stabilnym algorytmem porządkowania**. Zatem jeśli algorytm stabilny zastosujemy do uporządkowania alfabetycznej listy uczniów według innego kryterium lub innej cechy, to na otrzymanej liście uczniowie o tej samej wartości cechy porządkującej znajdą się w porządku alfabetycznym.

Ćwiczenie 10.8. Który z poznanych przez Ciebie algorytmów porządkowania jest stabilny, a który nie jest? ■

To, czy algorytm porządkujący jest stabilny, czy nie jest, może zależeć od sposobu jego realizacji (o tym traktują problemy 10.4 i 13.5).

Porządkowanie w miejscu (in situ)

Omawiając algorytm porządkowania przez scalanie oraz sposób jego realizacji, uwzględniliśmy, że ciąg tworzony w wyniku scalania dwóch innych ciągów nie może być natychmiast zapisany w tym samym miejscu (na papierze, w pamięci

komputera), w którym dane były ciągi do scalenia — nie jest to więc algorytm działający „w miejscu” — jest tutaj używany termin łaciński *in situ*.

Ćwiczenie 10.9. Przekonaj się, że wszystkie algorytmy porządkowania liczb, oprócz porządkowania przez scalanie, są algorytmami działającymi *in situ*. A co możesz powiedzieć o algorytmach: kubełkowym i pozycyjnym? ■

Ta własność algorytmów porządkowania ma duże znaczenie w praktycznych obliczeniach. Algorytmem działającym w miejscu można bowiem porządkować w tej samej pamięci operacyjnej dwa razy dłuższe ciągi niż algorytmem porządkującym przez scalanie, który wymaga z kolei dwa razy większej pamięci operacyjnej.



Ta wada algorytmu porządkowania przez scalanie przestaje być uciążliwa, gdy chcemy porządkować bardzo długie ciągi danych, które nie mieszczą się w pamięci operacyjnej komputera. W takiej sytuacji ciąg do uporządkowania może zajmować wiele dysków lub innych nośników danych.

Metoda porządkowania przez scalanie jest wówczas stosowana wielokrotnie do porządkowania ciągów wprowadzanych do komputera z zewnątrz, przy czym scalane ciągi są przeglądane liniowo od ich początków. Dokładny opis tego algorytmu można znaleźć np. w książce [Knuth-3]. ■

Złożoność i efektywność algorytmów porządkowania

Ze względu na złożoność obliczeniową (czyli liczbę operacji wykonywanych na elementach) algorytmy porządkowania liczb można zaliczyć do dwóch grup:

1. Algorytmy, w których liczba porównań jest proporcjonalna do kwadratu liczby elementów, czyli do n^2 . Są to algorytmy porządkowania bąbelkowego, przez wybór, przez umieszczanie i algorytm szybki. Zaliczenie dwóch pierwszych algorytmów do tej grupy nie budzi wątpliwości (zobacz np. ćwiczenie 6.3 i punkt 6.2). Algorytm porządkowania przez umieszczanie znalazł się w tej grupie, chociaż liczba porównań w nim wykonywanych jest mniejsza od n^2 , ale liczba przestawień elementów może być właśnie tego rzędu (ćwiczenie 10.2). Z kolei szybki algorytm porządkowania jest również w tej grupie, gdyż w najgorszym przypadku danych (zobacz problem 10.3) wykonuje się w nim liczbę porównań proporcjonalną do n^2 .

2. Algorytmy, w których liczba porównań i zamian jest proporcjonalna do $n \log_2 n$. Z tej grupy podaliśmy tylko jeden algorytm — porządkowanie przez scalanie (punkt 10.2.2).

Odrębną grupę stanowią algorytmy kubełkowy i pozycyjny oraz algorytmy

zasugerowane w problemach 6.2 i 13.25. Ich złożoność nie wyraża się liczbą porównań, gdyż nie są w nich wykonywane żadne porównania między porządkowanymi elementami. Działają one najbardziej efektywnie, gdy podane elementy mają wartości z przedziału o niewielkiej rozpiętości.

Złożoność obliczeniowa algorytmów jest często dość zgrubnym określeniem liczby wykonywanych w nich operacji — wielokrotnie sami piszemy, że liczba operacji jest proporcjonalna na przykład do n^2 , ale nie podajemy, w jakim stopniu. Posłużmy się konkretnym przykładem trzech algorytmów porządkowania: bąbelkowego, przez wybór i szybkiego. Liczba wykonywanych w nich porównań dla najgorszego układu danych jest proporcjonalna właśnie do n^2 , ale z tego nie wynika jeszcze, która z tych metod jest rzeczywiście najszybsza. Wyprzedziliśmy podanie odpowiedzi na to pytanie, nazywając jeden z tych algorytmów szybkim. A wśród dwóch pozostałych który jest szybszy?

Teoretyczna, czyli wyliczona na papierze, liczba działań w algorytmach porządkowania nie zawsze odpowiada praktycznej efektywności algorytmów. Jeśli potrafisz programować, możesz wykonać eksperymenty obliczeniowe z algorytmami porządkowania (zobacz problemy 10.6, 10.7 oraz 13.13). Jeśli nie potrafisz, posłuż się odpowiednim oprogramowaniem edukacyjnym.



Z demonstracją działania i z porównaniem efektywności wielu algorytmów porządkowania można się zapoznać, posługując się aplikacją *Sortowanie z [Oprog]*. Zajrzyj również do serwisu sieciowego, towarzyszącego tej książce — znajduje się tam moduł w języku Pascal, zawierający najważniejsze algorytmy porządkowania. ■

Złożoność problemu porządkowania

Rozważania w poprzednim punkcie dotyczą złożoności poszczególnych algorytmów porządkowania i ich praktycznej efektywności. Nasuwa się jednak wiele dalszych pytań — jedno z nich jest bardzo ogólne i dotyczy **złożoności problemu porządkowania**. Wyjaśnimy to dokładniej.

Podaliśmy kilka różnych algorytmów rozwiązywania problemu porządkowania, o różnej złożoności i różnej efektywności. Jesteśmy więc przygotowani do tego, by dla konkretnych danych do uporządkowania wybrać najbardziej odpowiednią metodę wśród poznanych. Chcielibyśmy jednak wiedzieć, czy dysponujemy algorytmami najlepszymi z możliwych. Dla kilku problemów w tej książce podaliśmy algorytmy optymalne, czyli wykonujące najmniejszą możliwą liczbę operacji. A jak jest z algorytmami porządkowania? Czy nie jest możliwe opracowanie jeszcze lepszego algorytmu niż te, które znamy? W tym właśnie sensie pytamy tutaj o złożoność problemu porządkowania i chcemy na nie

odpowiedzieć.

Odpowiedź jest tylko częściowo pozytywna: znamy bowiem najlepszy, ale tylko w pewnym sensie, algorytm porządkowania. Uzasadnienie tego stwierdzenia jest bardzo podobne do uzasadnienia podanego w punkcie 9.2, że algorytm binarnego umieszczania jest optymalny. W tym celu skorzystamy z interpretacji algorytmów w postaci drzew algorytmów. Przypomnijmy (zobacz punkt 4.1), że jeśli modelem algorytmu jest drzewo, to jego liście muszą zawierać wszystkie możliwe wyniki, jakie można otrzymać w danym problemie — wtedy wysokość takiego drzewa jest złożonością algorytmu w sensie pesymistycznym, tzn. jest największą liczbą operacji wykonywanych w algorytmie dla dowolnych danych. Określmy więc strukturę i wielkość drzewa dowolnego algorytmu porządkowania.

Liczba wszystkich możliwych wyników algorytmu porządkowania, zastosowanego do ciągu o długości n , jest równa liczbie wszystkich możliwych uporządkowań n liczb. Jest ona zatem równa $n!$ (zobacz punkt 1.2, rozdział 4., zadanie 8.3 i problem 13.2), czyli drzewo jakiegokolwiek algorytmu porządkowania musi zawierać przynajmniej $n!$ liści.

Ćwiczenie 10.10. Przypomnij sobie rozwiązania ćwiczeń z rozdziału 4. i przekonaj się, że utworzone tam drzewa algorytmów porządkujących kilka liczb potwierdzają prawdziwość stwierdzenia z poprzedniego akapitu. ■

Teraz, aby móc określić, jaki jest możliwie najlepszy algorytm porządkowania, musimy znaleźć, jakie jest najniższe drzewo, które zawiera przynajmniej $n!$ wierzchołków końcowych. Skorzystamy tutaj również z wielokrotnie przytaczanego faktu, że drzewo o wysokości k (lub inaczej drzewo o k poziomach) zawiera 2^k wierzchołków końcowych. Stąd wynika, że najmniejsza wysokość k drzewa algorytmu porządkowania n liczb spełnia nierówność:

$$n! \leq 2^k,$$

z której po zlogarytmowaniu obu stron otrzymujemy nierówność:

$$\log_2 n! \leq k.$$

Możemy jeszcze wykorzystać wzór Stirlinga na przybliżoną wartość $n!$. Otrzymamy wtedy, że wyrażenie $\log_2 n!$, stojące po lewej stronie ostatniej nierówności, jest równe w przybliżeniu $n \log_2 n$.

Z powyższego rozumowania można wyprowadzić wniosek, że nie jest możliwe opracowanie algorytmu porządkowania, w którym byłoby wykonywanych mniej niż około $n \log_2 n$ porównań. „Okolo” znaczy tutaj, że nie uwzględniamy w rozważaniach pozostałych członów w wyrażeniu na złożoność algorytmu, które mają mniejszy wpływ na liczbę działań — np. we wzorze (10.2) na złożoność algorytmu porządkowania przez scalanie występuje jeszcze składnik równy n

+ 1.

Ćwiczenie 10.11. Oblicz wartości $\log_2 n!$ dla $n = 2, 3, 4$ i 5 i porównaj je ze złożonościami algorytmów porządkowania $2, 3, 4$ i 5 liczb, przedstawionymi w rozdziale 4. ■

Poznaliśmy algorytm porządkowania, dla którego wzór na złożoność zawiera największy składnik równy $n \log_2 n$, czyli algorytm porządkowania przez scalanie (punkt 10.2.2). Zatem jest on **optymalny** pod względem złożoności obliczeniowej, musimy jednak dodać, że ze względu na inne człony występujące we wzorze na złożoność, jest to optymalność **z dokładnością do stałego współczynnika proporcjonalności**. Ale czy algorytm jest rzeczywiście najszybszy?

Ćwiczenie 10.12. Porównaj dla pierwszych pięciu potęg liczby 2 wartości wyrażeń: $\log_2 n!$ — oszacowania minimalnej liczby porównań w jakimkolwiek algorytmie porządkującym, $n \log_2 n$ — przybliżenie tego pierwszego wyrażenia i $n \log_2 n - n + 1$ — złożoność najlepszego znanego Ci algorytmu porządkowania. ■

10.5. Zadania i problemy

Zadanie 10.3. Podany algorytm porządkowania przez umieszczanie najpierw znajduje miejsce dla kolejnego elementu (krok 2.), a później „rozsuwa” ciąg, by wstawić ten element (w kroku 3.). Gdy to miejsce jest znajdowane za pomocą liniowego algorytmu umieszczania, wówczas te dwa kroki można połączyć w jeden, w którym podczas szukania miejsca elementy są przestawiane. Podaj opis takiego algorytmu i przedstaw jego realizację w wybranym przez siebie języku.

Zadanie 10.4. Podaj opis iteracyjnej wersji algorytmu porządkowania przez scalanie dla ciągów, których długość n jest potęgą liczby 2.

Wskazówka. Iteracyjny algorytm porządkowania przez scalanie można wyprowadzić, przyglądając się uważnie schematowi działania algorytmu rekurencyjnego pokazanemu na rysunku 10.3, przy czym główną uwagę należy skupić na etapie scalania. Zauważ, że jeśli n jest potęgą liczby 2, to najpierw są scalane ciągi o długości 1 w ciągi o długości 2, następnie te ciągi o długości 2 są scalane w ciągi o długości 4 itd. Kontynuacją tego zadania jest problem 13.12.

Problem 10.1. Wykaż, stosując metodę wstawiania (np. taką, jak podano w punkcie 8.2.1 i w punkcie 9.1), że rozwiązaniem zależności rekurencyjnej (10.1) dla n będącego potęgą liczby 2 (np. $n = 2^k$) jest równość (10.2).

Wskazówka. Szczegółowe wyprowadzenie tej równości znajduje się w podręczniku [EI-I].

Problem 10.2. W podanych opisach szybkiego algorytmu porządkowania w językach Pascal i Python elementem dzielącym ciąg jest element znajdujący się w połowie ciągu. Ten wybór ma pewne zalety. Rozważ dwie realizacje szybkiej metody porządkowania: jednej — z pierwszym elementem dzielącym, i drugiej — ze środkowym. Porównaj efektywność obu realizacji na ciągach: już uporządkowanym, uporządkowanym w odwrotnej kolejności oraz złożonym z jednakowych elementów.

Problem 10.3. Podaj odpowiednie przykłady ciągów elementów uzasadniające, że w obu wersjach algorytmu szybkiego porządkowania, z pierwszym i środkowym elementem dzielącym, liczba wykonywanych porównań jest proporcjonalna do n^2 , gdzie n jest liczbą elementów w ciągu.

Ceną, jaką płacimy za uniwersalność algorytmów porządkowania, są działania wykonywane przez nie nawet wtedy, gdy ciąg jest uporządkowany.

Zadanie 10.5. Który z poznanych przez Ciebie algorytmów porządkowania wykonuje najmniej działań na ciągu już uporządkowanym? Czy któryś algorytm nie wykonuje na takim ciągu żadnych działań? Przecież ciąg jest już uporządkowany!

Zadanie 10.6. Powtórz poprzednie zadanie dla ciągów danych w odwrotnej do poszukiwanej kolejności. Który z algorytmów wykonuje najmniej operacji na takich ciągach, a który — najwięcej?

Problem 10.4. Pokaż, że algorytm porządkowania przez scalanie można zrealizować zarówno jako stabilny, jak i niestabilny.

Problem 10.5. Sprawdź na przykładach, że bez względu na wybór elementu, którym dzielimy ciąg w szybkim algorytmie porządkowania na dwa podciągi, ten algorytm nie jest stabilny.

Problem 10.6. Wybierz dwa algorytmy porządkowania, których złożoność jest proporcjonalna do n^2 i porównaj ich praktyczną efektywność na ciągach o różnej długości, generowanych losowo. Porównaj również efektywność działania wybranych algorytmów na ciągach uporządkowanych i na ciągach w odwrotnej kolejności. Możesz posłużyć się zamieszczonymi tutaj opisami tych algorytmów w wybranym języku programowania lub skorzystać z możliwości porównania algorytmów w aplikacji *Sortowanie* [Oprog].

Problem 10.7. Wykonaj testowanie z poprzedniego problemu dla wszystkich poznanych w tej książce algorytmów porządkowania o złożoności proporcjonalnej do n^2 .

Mogłeś dowiedzieć się:

► jak **porządkować ciągi, porządkując ich podciągi**;
oraz poznać algorytmy **działające na zasadzie dziel i zwyciężaj**,

zastosowane do rozwiązywania następujących problemów:

- ▶ **scalania ciągów uporządkowanych** i użycia tej operacji do porządkowania dowolnych ciągów;
- ▶ porządkowania metodą, która w praktyce jest **najszybszą metodą porządkowania**.

Mogłeś również uporządkować swoje wiadomości o porządkowaniu.

Rozdział 11. Wychodzenie z labiryntu i pakowanie plecaka

Tu poznasz następujące problemy:

- ▶ **znajdowanie drogi do wyjścia z labiryntu**, w tym możliwie najkrótszej;
- ▶ **kompletowanie** najcenniejszej **zawartości plecaka**;

do których rozwiązywania są stosowane dwie nowe metody:

- ▶ **zachłanne heurystyki**;
- ▶ **programowanie dynamiczne**.

W tym rozdziale omawiamy dwa problemy i algorytmy ich rozwiązywania, które ilustrują bardziej złożone metody algorytmiki. Jeden z problemów polega na znalezieniu wyjścia z labiryntu, a drugi — na spakowaniu plecaka. W pierwszym — szukamy w labiryncie możliwie najkrótszej drogi do wyjścia, a w drugim — staramy się umieścić w plecaku możliwie najwięcej wartościowych rzeczy, gdy nie możemy zapakować wszystkich.

Heurystyka jest gałęzią nauki zajmującą się badaniem metod i reguł dokonywania odkryć w trakcie rozwiązywania problemów. Zapoczątkował ją Euklides, a później zajmowali się nią R. Descartes, G. W. Leibniz i B. Bolzano. Ostatnio heurystyką określa się najczęściej metodę rozwiązywania problemu, która niekoniecznie spełnia wszystkie warunki nakładane na algorytm — na przykład nie można powiedzieć, dla jakich danych wyznacza ona rozwiązania optymalne.

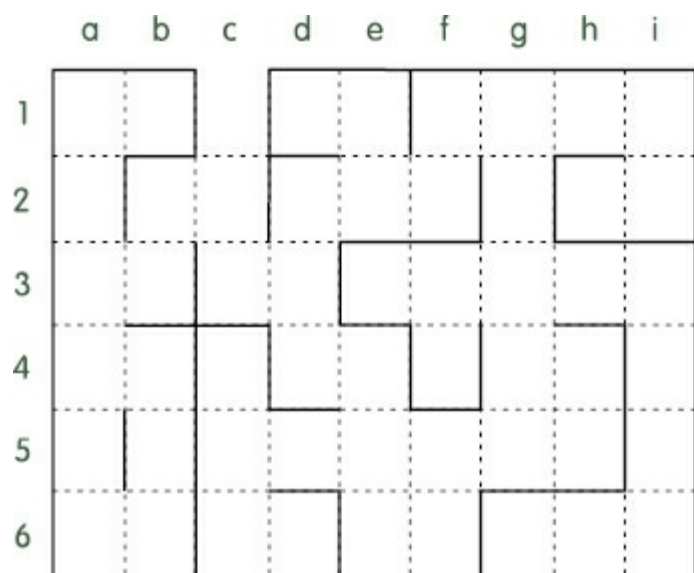
Problemy te należą do rodziny tak zwanych **problemów optymalizacji**, które dotyczą znajdowania najlepszego rozwiązania wśród wielu możliwych rozwiązań spełniających pewne warunki.

Nasze próby rozwiązania tych dwóch problemów są okazją do poznania dość ogólnych metod algorytmicznych, które mają znacznie szersze zastosowania niż tylko te tutaj przedstawione. Należą do nich **metody heurystyczne**, w których wykorzystuje się intuicyjne sposoby otrzymania możliwie najlepszych rozwiązań. Nie zawsze je osiągamy. Są to jednak metody na tyle szybkie, że mają praktyczne znaczenie.

11.1. Znajdowanie wyjścia z labiryntu

Zakładamy, że **labirynt** jest zamknięty w prostokącie, ma tylko jedno wyjście

(wejście) i wszystkie jego ściany wewnętrzne są równoległe do zewnętrznych. Dla uproszczenia rozważań założmy również, że ściany są fragmentami siatki kwadratowej. Zatem wnętrze labiryntu jest złożone z kwadratowych pól tej siatki i każdy wewnętrzny punkt labiryntu możemy utożsamiać z polem, w którym leży. Ponadto przyjmujemy, że w labiryncie nie ma zamkniętych komnat, a więc z każdego punktu wewnętrznego istnieje droga prowadząca do wyjścia (rysunek 11.1).



Rysunek 11.1. Przykładowy labirynt

Problem znalezienia drogi w labiryncie, a dokładniej — wyjścia z labiryntu, jest tak stary, jak grecka mitologia: Tezeusz miał odnaleźć w labiryncie potwora Minotaura i po jego zabiciu wyjść z labiryntu — w tym miała mu pomóc nić podarowana przez Ariadnę.

Naszym celem jest podanie algorytmu, który z każdego punktu labiryntu zaprowadzi nas do jego wyjścia, bez zbędnego kluczenia^[1].

Zacznijmy od ścisłego opisu tego problemu w postaci specyfikacji:

Problem znalezienia wyjścia z labiryntu

Dane: Labirynt, czyli prostokąt z jednym wyjściem, wypełniony ścianami, które są równoległe do zewnętrznych ścian i nie tworzą zamkniętych obszarów. Dany jest również punkt (pole) s wewnątrz labiryntu.

Wynik: Droga w labiryncie złożona z ciągu pól, która prowadzi z punktu (pola) s do wyjścia. ■

Metoda wychodzenia z labiryntu jest na ogół opisem sposobu chodzenia po istniejących odcinkach korytarzy (utworzonych w naszym przypadku z pól) i można w niej wyróżnić dwa elementy: regułę gwarantującą, że żadnego odcinka drogi w labiryncie nie przechodzimy więcej niż jeden raz, i strategię jak najszybszego znalezienia wyjścia z labiryntu. W następnych punktach opiszemy

trzy metody, które mają cechy postępowania: naiwnego, zachłannego i optymalnego — dokładniejszy sens tych określeń stanie się jasny po zapoznaniu się z tymi metodami.

Zaznaczmy tutaj jeszcze, że dwie pierwsze metody mogą być stosowane w sytuacji, gdy znajdujemy się w labiryncie i nie znamy jego schematu, możemy więc korzystać jedynie z lokalnych informacji, które jesteśmy w stanie zgromadzić, rozglądając się wokół siebie. W trzeciej metodzie korzystamy natomiast z mapy całego labiryntu, nadaje się więc ona do wyznaczania dróg z wyprzedzeniem.

11.1.1. Wychodzenie z labiryntu po omacku

Pierwsza metoda ma cechy bardzo „nerwowej” strategii i zapewne stosowalibyśmy ją, gdybyśmy nagle znaleźli się w labiryncie i starali się po ciemku trafić do wyjścia. Wtedy po wybraniu kierunku poruszania się, trzymając cały czas jedną (ale tę samą) rękę na ścianie, podążalibyśmy wzdłuż ścian. Można więc tę metodę nazwać **metodą z ręką na ścianie**. Jest oczywiste, że — poruszając się w ten sposób — albo trafimy do wyjścia, albo wrócimy do punktu, w którym już byliśmy.

Ćwiczenie 11.1. Posługując się metodą z ręką na ścianie, spróbuj znaleźć drogę do wyjścia z labiryntu przedstawionego na rysunku 11.1, gdy znajdujesz się w punkcie 3b oraz w punkcie 4f. Jaki wpływ na trasę Twojej wędrówki będzie miała zamiana ręki i zmiana kierunku poruszania się wzdłuż ścian? ■

Zadanie 11.1. Jakie warunki musi spełniać labirynt, aby z każdego jego punktu można było dojść do wyjścia, trzymając rękę na ścianie? Uzasadnij swoją odpowiedź. ■

Nie będziemy więcej zajmować się tą metodą, gdyż poza naturalnością sformułowania nie ma ona ciekawych własności.

11.1.2. Przeszukiwanie z nawrotami w poszukiwaniu wyjścia

Omówimy tutaj bardziej systematyczną metodę wychodzenia z labiryntu.

W każdym punkcie (polu) labiryntu są co najwyżej cztery możliwości wykonania następnego kroku: {w górę, w lewo, w prawo, w dół} — oznaczmy te kierunki przez (G, L, P, D). Możemy więc zaproponować następującą zasadę poruszania się: w danym polu, w którym się znalazłeś, wybierz z listy kierunków pierwszy jeszcze niezbadany kierunek przejścia z tego pola, w którym istnieje pole nieoddzielone ścianą i jeszcze dotychczas na nim nie byłeś, i przejdź na to pole. Natomiast jeśli z danego pola nie można już przejść w żadnym kierunku, to wróć

do pola, z którego przyszedłeś, i kontynuuj to postępowanie. Krok będący powrotem oznaczmy literą B. Każdy ruch możemy opisać nazwą kroku i nazwą pola, na które nas wiedzie. Zróbmy jeszcze jedno, dość naturalne założenie: kierunek poruszania się po labiryncie określamy w zależności od naszego ustawienia i przyjmujemy przy tym, że cały czas poruszamy się twarzą do przodu, z wyjątkiem ruchów typu B. Stąd wynika, że kierunek G jest zawsze przed nami.

Pierwowzór opisanego w tym punkcie algorytmu pojawił się ponad sto lat temu. W różnych wersjach podali go L. C. Wiener, M. Trémaux i G. Tarry.

Zastosujmy powyższą zasadę do znalezienia wyjścia z pola 3b w labiryncie pokazanym na rysunku 11.1. Pierwszy ruch ma postać G-2b, ale w następnym kroku nie możemy już iść ani do góry, ani w lewo, więc wykonujemy ruch P-2c, a z pola 2c — ruch L-1c i już jesteśmy przy wyjściu z labiryntu.

Jeśli wyjścia szukamy z pola 4a, to początkowy fragment drogi ma postać: G-3a, G-2a, G-1a. Z pola 1a nie możemy iść ani w kierunku G, ani L, wykonujemy więc ruch P-1b. Z pola 1b nie możemy już przejść do żadnego nowego pola, wracamy więc i to aż do pola 3a: B-1a, B-2a, B-3a. Z pola 3a możemy teraz przejść w prawo, a więc wykonujemy kolejne ruchy: P-3b, L-2b, P-2c, L-1c.

Jeśli poszukiwanie wyjścia rozpoczynamy w punkcie 2a, to początkowe ruchy są podobne: G-1a, P-1b, B-1a, B-2a i wracamy do punktu wyjścia. Możliwy do wykonania pozostał jeszcze ruch do dołu: D-3a. Znaleźliśmy się w punkcie, z którego poprzednie poszukiwanie wyjścia zakończyło się dość szybko. Teraz jednak nasz kierunek poruszania się jest do dołu (zgodnie z zasadą „twarzą do przodu”), więc następnymi ruchami są: G-4a, G-5a, G-6a, L-6b, L-5b, G-4b. Z pola 4b nie możemy jednak przejść na pole 4a, aby się nie zapętlić, gdyż przechodziliśmy już przez nie w tym poszukiwaniu wyjścia, zatem wracamy: B-5b, B-6b, B-6a, B-5a, B-4a, B-3a, a dalej już do wyjścia: L-3b, L-2b, P-2c, L-1c.

Ćwiczenie 11.2. Posługując się powyższym algorytmem, znajdź drogę do wyjścia z pola 4g w labiryncie pokazanym na rysunku 11.1. Jest to jednak dość długa droga — aby ją znaleźć, musisz przejść przez 26 pól, wielokrotnie się cofając. ■

Przedstawiony algorytm jest realizacją metody przeszukiwania przestrzeni możliwych rozwiązań, która nazywa się **przeszukiwaniem w głąb**, gdyż w kolejnych krokach przeszukiwanie zagłębia się coraz bardziej — tak daleko, jak to możliwe. Tę metodę można również nazwać **przeszukiwaniem z nawrotami**, gdyż są w niej wykonywane nawroty do poprzednich punktów, gdy poszukiwanie zabrze w ślepy zaułek. O poszukiwaniu z nawrotami piszemy więcej w książce [Piramidy], w rozdziale 14.

Metoda przeszukiwania w głąb jest jedną z najważniejszych metod konstruowania algorytmów w teorii grafów. **Grafy** są popularnymi modelami dla zadań znajdowania dróg o różnych własnościach, zobacz w książce [Sysło],

patrz także [Wilson].

Opiszemy teraz ściśle algorytm wychodzenia z labiryntu w postaci rekurencyjnej. Zakładamy, że na początku w żadnym polu labiryntu jeszcze nie byliśmy, wszystkie pola są więc **nieodwiedzone**.

Algorytm znajdowania wyjścia z labiryntu (v)

Dane: Labirynt, czyli prostokąt z jednym wyjściem, wypełniony ścianami, które są równoległe do zewnętrznych ścian i nie tworzą zamkniętych obszarów. Dany jest również punkt v wewnątrz labiryntu.

Wynik: Droga w labiryncie, która prowadzi z punktu v do wyjścia.

Krok 1. Dla każdego kolejnego kierunku (G, L, P, D) poruszania się z pola v , jeśli istnieje w tym kierunku nieodwiedzone pole w i nie jest ono odgródzone od pola v ścianą, to przejdź do kroku 2., a w przeciwnym razie — zakończ to wywołanie tego algorytmu.

Krok 2 Jeśli wyjście z labiryntu jest w jednej ze ścian pola w , to zakończ algorytm. W przeciwnym razie oznacz pole w jako odwiedzone i wywołaj ten algorytm dla tego pola w . ■

Zauważmy, jak proste jest sformułowanie tego algorytmu. Tę prostotę zawdzięczamy przede wszystkim użyciu w jego opisie rekurencji (porównaj ten opis z iteracyjną realizacją tego algorytmu w rozwiązaniu problemu 13.16).

Ćwiczenie 11.3. Wyjaśnij, dlaczego w powyższym opisie nie występuje jawnie ruch, który w słownym opisie algorytmu jest oznaczony literą B. Czy rzeczywiście taki ruch nie jest wykonywany w tym algorytmie? ■

Ćwiczenie 11.4. Zastosuj algorytm z nawrotami do znalezienia wyjścia z labiryntu przedstawionego na rysunku 11.1, z pól $v = 6b$ i $v = 4c$. ■

Zgodnie z naszym założeniem, w labiryncie nie ma obszarów zamkniętych, a więc z każdego pola istnieje droga do wyjścia. Na tej podstawie nie powinienś mieć kłopotu z wykonaniem następnego ćwiczenia.

Ćwiczenie 11.5. Uzasadnij, że dla każdego pola v algorytm z nawrotami znajdzie drogę z tego pola do wyjścia z labiryntu. ■

11.1.3. Znajdowanie najkrótszej drogi do wyjścia z labiryntu

Rób lepiej to, co inni robią dobrze.

Oreste Vaccari

Wynik ćwiczenia 11.5 jest dla nas zadowalający w tym sensie, że jeśli z danego

pola w labiryncie istnieje droga do wyjścia, to posługując się algorytmem z nawrotami, zawsze ją znajdziemy. Nie może nas jednak satysfakcjonować szybkość wykonania tego zadania — długo trzeba krążyć, aby trafić do wyjścia. Jest jeszcze inny powód, dla którego nie powinniśmy być zadowoleni z wyników działania tego algorytmu.

Ćwiczenie 11.6. Znajdź drogę z pola 4g do wyjścia labiryntu pokazanego na rysunku 11.1, stosując algorytm przeszukiwania z nawrotami. Przez ile pól wiedzie znaleziona droga? Znajdź w tym labiryncie drogę z pola 4g do wyjścia, przechodzącą przez mniejszą liczbę pól. ■

To ćwiczenie ma pokazać, że algorytm przeszukiwania z nawrotami nie zapewnia znalezienia drogi, która prowadzi do wyjścia przez najmniejszą liczbę pól — nazwalibyśmy ją **najkrótszą drogą**. A jak to osiągnąć? Jeśli z danego pola można przejść do wyjścia wieloma różnymi drogami, to jedna z metod może polegać na wygenerowaniu wszystkich takich dróg i wybraniu najkrótszej. Ale takich dróg jest na ogół dużo, chcielibyśmy więc dysponować algorytmem, według którego z ustalonego pola niemal bezpośrednio podążamy do wyjścia najkrótszą drogą. Jednego jesteśmy pewni — taka najkrótsza droga z wybranego pola do wyjścia zawsze istnieje, ponieważ liczba wszystkich dróg z każdego pola do wyjścia jest skończona.

Obok uzasadniamy bardzo pożyteczną własność najkrótszych dróg wiodących do celu: każdy fragment takiej drogi między dwoma dowolnymi jej punktami jest również najkrótszą drogą między tymi punktami.

Spójrzmy na poszukiwaną najkrótszą drogę od jej końca, czyli od wyjścia. Jeśli składa się ona z l pól, to ostatnie pole na tej drodze jest poprzedzone przez $l - 1$ pól, przedostatnie — jest poprzedzone przez $l - 2$ pól itd. Co więcej, jeśli w tej drodze staniemy na jakimkolwiek jej polu r , to liczba pól tej drogi między początkowym polem s a polem r jest również możliwie najmniejsza, ponieważ gdyby istniała w labiryncie krótsza droga między tymi polami, wówczas moglibyśmy nią pójść, by skrócić sobie trasę z pola s do wyjścia.

Te obserwacje sugerują, że w poszukiwaniu najkrótszej drogi z wybranego pola s do wyjścia powinniśmy na każdym kroku poruszać się po najkrótszej drodze z pola s . W jaki sposób możemy to zapewnić? Istnieje prosta strategia, wynikająca również z tej dyskusji: najpierw generujemy pola, które są w odległości jednego pola od s , potem pola, które są w odległości dwóch pól od s i tak dalej. Stąd widać, że jeśli wyjście jest w odległości l pól od s , to dotrzemy do niego po l krokach, po drodze, która się składa z l pól. Ale w jaki sposób znajdować pola odległe o jedno, dwa, ... l pól od s ? To już jest proste na podstawie dyskusji w poprzednim akapicie — pole odległe o dwa pola od s jest przedzielone polem, które przylega do s , pole odległe o trzy pola od s przylega do pola odległego o dwa pola od s itd. Zatem w algorytmie będziemy generowali kolejno: pola odległe o jedno pole od s (czyli przyległe do s), pola odległe o dwa pola od s ,

którymi są pola przyległe do pól odległych o jedno pole od s itd. Taką metodę generowania ciągu pól można nazwać algorytmem **bliższe-najpierw**, gdyż najpierw wybieramy pola, które są bliższe miejscu, gdzie stoimy.

Ćwiczenie 11.7. Sprawdź, że stosując opisany algorytm w poszukiwaniu najkrótszej drogi z pola 4g do wyjścia z labiryntu pokazanego na rysunku 11.1, przejdziesz przez wszystkie pola oznaczone liczbami na rysunku 11.2. Liczby te są długościami dróg (w sensie liczby pól) z pola s do tych pól. ■

	a	b	c	d	e	f	g	h	i
1			9	8	7	4	3	4	5
2		9	8	7	6	5	2	7	6
3			7	6	3	2	1	2	3
4			6	5	4	3	s	1	4
5			5	4	3	2	1	2	5
6			6	7	4	3	8	7	6

Rysunek 11.2. Numeracja pól labiryntu przedstawionego na rysunku 11.1 w kolejności bliższe-najpierw względem pola s

Algorytm zastosowany do labiryntu wypełnia go liczbami aż do osiągnięcia pola przyległego do wyjścia. Ale jak z tych liczb można odczytać pola tworzące najkrótszą drogę? W tym celu ponownie korzystamy z własności wyprowadzonej wyżej, że każdy fragment najkrótszej drogi jest również najkrótszą drogą. Zatem jeśli droga z pola s do pola przyległego do wyjścia składa się z 9 pól (tak, jak na rysunku 11.2), to poprzednie pole na tej drodze powinno być odległe od s o 8 pól, czyli powinna być w nim liczba 8 — jest to pole 2c. Na tej samej zasadzie poprzedzającym go polem jest 3c, a dla 3c — jest pole 3d. Do pola 3d, w którym jest liczba 6, są przyległe pola 2d i 4d — wybieramy pole 4d, gdyż jest w nim liczba 5 o jeden mniejsza od 6. Postępujemy tak aż do osiągnięcia pola s. Przebytą w ten sposób drogę oznaczyliśmy na rysunku 11.2 pogrubieniem.

Ćwiczenie 11.8. Porównaj liczbę pól odwiedzonych w trakcie działania algorytmu przeszukiwania z nawrotami i algorytmu znajdowania najkrótszej drogi dla przykładu z poprzednich ćwiczeń. W tym pierwszym algorytmie uwzględnij również pola, które są odwiedzane w ruchu powrotnym. ■

Opisany algorytm znajdowania najkrótszej drogi do wyjścia w labiryncie jest szczególnym przypadkiem **algorytmu Dijkstry** wyznaczania najkrótszej drogi w dowolnej sieci połączeń, w której odległości między punktami są nieujemne.

Wynik ostatniego ćwiczenia pokazuje, że liczba pól odwiedzanych podczas znajdowania najkrótszej drogi do wyjścia z labiryntu nie zawsze jest najmniejsza. Jednak nie są wykonywane ruchy, z których trzeba się wycofywać, jak w algorytmie z nawrotami. Ponad wszystko jest to poprawna metoda znajdowania najkrótszej drogi, nawet w bardziej złożonych sytuacjach.



Dowód tego stwierdzenia oraz opis innych zastosowań omówionego algorytmu znajdziesz w książce [Sysło]. ■

Aby przedstawić ścisły opis tego algorytmu, musimy jeszcze podać, w jaki sposób będziemy pamiętać kolejno odwiedzane i przeglądane pola. Podobnie jak w algorytmie przeszukiwania z nawrotami, zakładamy, że na początku algorytmu na żadnym polu jeszcze nie staliśmy, czyli wszystkie pola są nieodwiedzone. Aby mieć pewność, że pola przechodzimy w kolejności ich odległości od s , będziemy je umieszczali w kolejności osiągania, jedno po drugim w ciągu. W tej samej kolejności będą one opuszczać ten ciąg, gdy będziemy z nich przechodzić na nowe pola, leżące o jedno pole dalej od s . Co przypomina taki ciąg, w którym elementy znajdują się w kolejności napływania i w takiej samej kolejności opuszczają one ten ciąg? Jest to tradycyjna **kolejka**, w której nie obowiązują żadne przywileje — w algorytmie poniżej oznaczamy ją przez Q .

Algorytm znajdowania najkrótszej drogi do wyjścia z labiryntu

Dane: Labirynt, czyli prostokąt z jednym wyjściem, wypełniony ścianami, które są równoległe do zewnętrznych ścian i nie tworzą zamkniętych obszarów. Dany jest również punkt w polu s wewnątrz labiryntu.

Wynik: Droga w labiryncie, która prowadzi z pola s do wyjścia.

Krok 0. Przyjmij, że na początku nie byłeś w żadnym polu labiryntu, a więc wszystkie pola są nieodwiedzone.

Krok 1. Umieść w kolejce Q pole s . W polu s umieść liczbę 0.

Krok 2. Dopóki kolejka Q nie jest pusta, wykonuj kroki 3. – 5.

Krok 3. Usuń z kolejki Q jej pierwszy element — oznaczmy to pole przez v .

Krok 4. Dla każdego pola w sąsiedniego względem v i nieoddzielonego od niego ścianą wykonaj krok 5.

Krok 5. Jeśli nie byłeś jeszcze na polu w , to umieść w nim liczbę o jeden większą od liczby znajdującej się w polu v . Jeśli pole w zawiera wyjście, to przejdź do kroku 6., w przeciwnym razie dołącz w do końca kolejki Q .

Krok 6. {W tym kroku budujemy od końca listę pól tworzących najkrótszą drogę z pola s do pola w , na którym zakończył działanie krok 5.}

Dopóki w nie jest polem s , wykonuj: za kolejne (od końca) pole drogi przyjmij w i za nową wartość w przyjmij pole sąsiednie względem w , w którym znajduje się liczba o jeden mniejsza od liczby znajdującej się w obecnym polu w . ■

Ze względu na złożoność struktur danych, których należałoby użyć do reprezentowania labiryntu, przedstawiamy jedynie realizacje opisanych algorytmów w postaci procedur w języku Pascal.

George Pólya nazywa to **paradoksem odkrywcy**: często łatwiej jest podać rozwiązanie ogólniejszego problemu niż mniej ogólnego.

Algorytmy znajdowania wyjścia z labiryntu: przeszukiwania z nawrotami i znajdowania najkrótszej drogi do wyjścia, zostały zrealizowane w językach Pascal i Python, a ich teksty są dostępne na stronie towarzyszącej książce. Komputerowe realizacje tych algorytmów podajemy w ogólniejszej postaci, dzięki temu są... o wiele prostsze (zobacz komentarz w ramce obok). Działają one bowiem na dowolnych grafach. Zatem aby móc je zastosować do poszukiwania wyjścia z labiryntu, należy przygotować jedynie odpowiednie dane. Pozostawiamy to jako zadanie do samodzielnego wykonania dla tych, którzy znają bardziej zaawansowane typy danych (zobacz problem 11.2).

11.2. Pakowanie najcenniejszego plecaka

W tym punkcie zajmiemy się rozwiązywaniem różnych wersji **problemu plecakowego**. Polega on na zapakowaniu do plecaka o ograniczonej pojemności możliwie najbardziej wartościowych rzeczy. W innych zastosowaniach może to dotyczyć pakowania walizek, paczek, samochodu, samolotu itp. Opiszmy ogólną wersję tego problemu w postaci specyfikacji:

Ogólny problem plecakowy

Dane: n rzeczy (towarów, produktów itp.) R_1, R_2, \dots, R_n , każda w nieograniczonej ilości; rzecz R_i waży (lub zajmuje miejsce o wielkości) w_i jednostek i ma wartość p_i . Ponadto dana jest maksymalna pojemność plecaka, wynosząca W jednostek.

Wyniki: Ilości q_1, q_2, \dots, q_n , poszczególnych rzeczy (mogą być zerowe), których całkowita waga nie przekracza W i których sumaryczna wartość jest największa wśród wypełnień plecaka rzeczami o wadze nie przekraczającej W . ■

Problem plecakowy można sformułować w następującej postaci matematycznej: znaleźć wartości q_1, q_2, \dots, q_n , dla których wartość sumy będącej całkowitą wartością plecaka:

$$p_1 q_1 + p_2 q_2 + \dots + p_n q_n \quad (11.1)$$

jest największa, przy czym muszą być spełnione warunki, że nie będzie przekroczona pojemność plecaka, czyli:

$$w_1 q_1 + w_2 q_2 + \dots + w_n q_n \leq W \quad (11.2)$$

oraz wielkości q_1, q_2, \dots, q_n będą nieujemnymi liczbami całkowitymi (11.3)

Zwróć uwagę, że określenie „optymalny” odnosi się tutaj do rozwiązania problemu, podczas gdy dotychczas była mowa o algorytmach optymalnych, czyli wykonujących najmniejszą możliwą liczbę operacji.

Wartości q_1, q_2, \dots, q_n , które spełniają warunki (11.2) i (11.3), nazywamy **rozwiązaniem dopuszczalnym**, gdyż spełniają warunki specyfikacji problemu, a jeśli dodatkowo suma (11.1) dla tych wartości jest największa, to stanowią one **rozwiązanie optymalne**. ■

Tę wersję problemu plecakowego określamy jako **ogólną**, gdyż założyliśmy w specyfikacji, że dysponujemy nieograniczoną ilością każdej z rzeczy. Często jednak, pakując plecak, mamy tylko po jednej sztuce każdej rzeczy, wtedy nasz wybór polega jedynie na podjęciu decyzji: wziąć tę rzecz czy jej nie brać. Stąd wynika, że wielkość q_i może przyjąć w tym przypadku tylko wartość: 1 — gdy rzecz i dokładamy do plecaka, lub 0 — gdy jej nie bierzemy. Taką wersję tego problemu nazywamy **decyzyjnym problemem plecakowym**. Mimo ograniczenia możliwych wartości q_i , jest to dość ogólna wersja problemu plecakowego, gdyż — jeśli jakaś rzecz występuje w większej ilości, to możemy każdy jej egzemplarz nazwać inaczej i wtedy każda rzecz (egzemplarz) będzie występować pojedynczo.

Ćwiczenie 11.9. Sprawdź, że w sformułowaniu problemu plecakowego (w obu wersjach) można przyjąć, iż spełniony jest warunek $w_i \leq W$ dla każdego i . Ponadto w decyzyjnym problemie plecakowym można również założyć, że $w_1 + w_2 + \dots + w_n > W$. ■

Problem plecakowy, z pozoru bardzo prosty, jest dość trudny do rozwiązania. Opracowano do jego rozwiązywania wiele różnych algorytmów, które dobrze ilustrują metody rozwiązywania problemów optymalizacyjnych. Przykład tego problemu posłuży nam do przedstawienia i wyjaśnienia dwóch sposobów tworzenia algorytmów dla takich problemów. Pierwszy jest związany z działaniem „zachłannym”, prowadzącym do uzyskania możliwie najlepszego rozwiązania, a drugi — programowanie dynamiczne — jest systematyczną

metodą znajdowania optymalnych rozwiązań.



Szeroka dyskusja na temat przybliżonych i dokładnych metod rozwiązywania problemu plecakowego w różnych wersjach jest zamieszczona w książce [Sysło], rozdz. 2.1. ■

Wszystkie przedstawione w tym punkcie algorytmy rozwiązywania problemu plecakowego, ogólnego i decyzyjnego, zachłanne i wykorzystujące programowanie dynamiczne, są zrealizowane w języku Pascal w module Plecaki, dostępnym na stronie internetowej tej książki:

<http://edukacja.helion.pl/algorytmika>.

11.2.1. Algorytm zachłanny dla ogólnego problemu plecakowego

Przypomnijmy sobie, w jaki sposób na ogół pakujemy plecak. Dobrze, gdy wszystkie zaplanowane do wzięcia rzeczy mieszczą się w nim. Ale jeśli nie, to najczęściej działamy dość chaotycznie i w naszych decyzjach ścierają się ze sobą trzy kryteria wyboru kolejnych rzeczy do zapakowania:

- wybierać najcenniejsze rzeczy, czyli w kolejności nierosnących wartości p_i ;
- wybierać rzeczy zajmujące najmniej miejsca, czyli w kolejności niemalejących wag w_i ;
- wybierać rzeczy najcenniejsze w stosunku do swojej wagi, czyli w kolejności nierosnących wartości ilorazu p_i/w_i ; iloraz ten można uznać za jednostkową wartość rzeczy i .

Wszystkie te kryteria wyboru są przejawem strategii, którą można nazwać **działaniem zachłannym**. W takim postępowaniu zawartość plecaka jest kompletowana krok po kroku i każda decyzja polega na dokonaniu wyboru, który wydaje się być najlepszy na danym etapie, z oczekiwaniem, że ostatecznie doprowadzi to do znalezienia optymalnego rozwiązania. Ta strategia, jak pokażemy, niestety nie zawsze gwarantuje otrzymanie optymalnego rozwiązania.

Dla danych zamieszczonych w tabeli 11.1 wyznaczmy najcenniejsze zawartości plecaka w ogólnym problemie plecakowym (zobacz tabela 11.2), stosując każde z powyższych kryteriów wyboru kolejnej rzeczy.

Tabela 11.1. Dane do przykładu ogólnego problemu plecakowego

i 1 2 3 4 5 6 W

pi 6 4 5 6 10 2

wi 6 2 3 2 3 1 2 3

Pakując rzeczy w kolejności ich wartości, czyli w kolejności wartości współczynników p_i , najpierw wybieramy rzecz nr 5 i możemy wziąć jej aż 7 sztuk, gdyż każda z nich waży 3 jednostki, a pojemność plecaka wynosi 23. Pozostaje w plecaku miejsce na rzecz, która waży 2 jednostki. Następna w kolejności wartości jest rzecz nr 4 — waży ona akurat 2 jednostki, więc ją pakujemy. Zatem pakując plecak w kolejności wartości rzeczy, wybieramy 7 sztuk rzeczy nr 5 i jedną rzecz nr 4 — otrzymujemy zawartość plecaka o wartości 77.

Pakując rzeczy w kolejności wag, możemy umieścić 23 sztuki rzeczy najlżejszej — nr 6, ale ta zawartość plecaka ma zaledwie wartość tylko 46.

Pakujemy teraz plecak w kolejności nierosnących proporcji wartości do wagi. W naszym przykładzie nierosnącej kolejności tych ilorazów ($7/2$, $10/3$, $4/2$, $2/1$, $5/3$, $6/6$) odpowiada następująca kolejność rzeczy: (4, 5, 2, 6, 3, 1). Zatem najpierw umieszczamy w plecaku 11 sztuk rzeczy nr 4, z których każda waży 2 jednostki. Pozostałą jednostkę pojemności plecaka zapełniamy rzeczą nr 6. Otrzymujemy zawartość plecaka o wartości 79, najcenniejszą wśród zachłannie pakowanych.

Ćwiczenie 11.10. Nie powinieneś mieć większego kłopotu z zapakowaniem do plecaka rzeczy o łącznej wartości wynoszącej 80. ■

Wykazaliśmy na tym przykładzie, że żadna z trzech powyższych strategii zachłannych nie gwarantuje znalezienia najlepszego wypełnienia plecaka. Otrzymane **rozwiązania** są **przybliżone** w stosunku do rozwiązania optymalnego. Najbardziej uzasadnione jednak wydaje się być trzecie kryterium, gdyż są w nim uwzględnione oba parametry opisujące każdą rzecz: wartość i waga. Dlatego w dalszych rozważaniach ograniczamy uwagę jedynie do tego sposobu kompletowania zawartości plecaka, stosując metody zachłanne. Możemy więc założyć, że rzeczy przewidziane do zabrania w plecaku są uporządkowane zgodnie z tym kryterium wyboru, czyli mamy:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n} \quad (11.4)$$

Algorytm zachłanny dla ogólnego problemu plecakowego

Dane: n rzeczy, każda w nieograniczonej ilości, ważących w_i i mających wartość p_i ($i = 1, 2, \dots, n$); rzeczy zostały ponumerowane tak, że są spełnione nierówności (11.4). Ponadto dana jest maksymalna pojemność plecaka,

wynosząca W .

Wyniki: Ilości q_1, q_2, \dots, q_n poszczególnych rzeczy (mogą być zerowe), których całkowita waga nie przekracza W .

Krok 1. Dla kolejnych rzeczy $i = 1, 2, \dots, n$, uporządkowanych zgodnie z nierównościami (11.4), wykonaj krok 2.

Krok 2. Określ największą wartość q_i , spełniającą nierówność $w_i q_i \leq W$. Przyjmij $W := W - w_i q_i$.

Krok 3. Znaleziony ładunek plecaka ma wartość $p_1 q_1 + p_2 q_2 + \dots + p_n q_n$. ■

W punkcie 11.2.3 powrócimy do algorytmu zachłannego dla drugiej wersji problemu plecakowego — tam okaże się on bardziej przydatny, a jego niewielka modyfikacja da całkiem niezłe rozwiązania przybliżone (zobacz problem 11.4).

11.2.2. Programowanie dynamiczne dla ogólnego problemu plecakowego

Opiszemy tutaj metodę **programowania dynamicznego**, która zapewnia znalezienie optymalnego rozwiązania ogólnego problemu plecakowego. Pierwsze słowo w nazwie tej metody nie ma jednak wiele wspólnego z programowaniem, rozumianym jako ogół czynności, których celem jest opracowanie programu dla komputera, chociaż rozwiązania budowane tą metodą są często zapisywane w postaci programu komputerowego.

Podstawy programowania dynamicznego rozwinął Richard Bellman w latach 50. „Programowanie” w nazwie tej metody (podobnie jak w nazwach innych metod optymalizacji, np. programowanie liniowe) w czasach II wojny światowej i bezpośrednio po niej odnosiło się do „programowania”, czyli podejmowania najlepszych strategicznie decyzji. Metody matematyczne, jak programowanie dynamiczne czy liniowe, a w ogólności — programowanie matematyczne, stanowiły wtedy teoretyczne podstawy decyzji strategicznych. „Programowanie” odnosi się także do sposobu zapisywania rozwiązań w postaci tablicy.

Metodę programowania dynamicznego można uznać za pewne zastosowanie zasady dziel i zwyciężaj. Nie potrafimy jednak zawczasu przewidzieć, z których rozwiązań mniejszych problemów będziemy korzystać w następnych krokach i dlatego, przygotowując się na każdą ewentualność, rozwiązujemy wszystkie mniejsze problemy. W przypadku problemu plecakowego mniejsze problemy odpowiadają mniejszej liczbie rodzajów rzeczy, które do niego pakujemy i mniejszej pojemności plecaka. Rozwiązania wszystkich podproblemów można więc umieścić w tablicy, której wiersze odpowiadają kolejnym rzeczom, czyli są ponumerowane liczbami od 1 do n , a kolumny są kolejnymi pojemnościami plecaka, od 1 aż do maksymalnej dopuszczalnej, równej W (zobacz tabela 11.2).

Tabela 11.2. Tablica P wartości upakowań plecaka, wygenerowanych przez algorytm programowania dynamicznego zastosowany do rozwiązania przykładu ogólnego problemu plecakowego podanego w tabeli 11.1

j i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	0	0	6	6	6	6	6	6	6	12	12	12	12	12	12	18	18	18	18	18	18
2	0	4	4	8	8	12	12	16	16	20	20	24	24	28	28	32	32	36	36	40	40	44	44
3	0	4	5	8	9	12	13	16	17	20	21	24	25	28	29	32	33	36	37	40	41	44	45
4	0	7	7	14	14	21	21	28	28	35	35	42	42	49	49	56	56	63	63	70	70	77	77
5	0	7	10	14	17	21	24	28	31	35	38	42	45	49	52	56	59	63	66	70	73	77	80
6	2	7	10	14	17	21	24	28	31	35	38	42	45	49	52	56	59	63	66	70	73	77	80

Pola w tej tablicy oznaczmy przez $P_{i,j}$, gdzie i jest numerem (indeksem) wiersza i przebiega wszystkie rodzaje rzeczy, które mogą być umieszczone w plecaku. Kolejność rozpatrywania rzeczy nie ma znaczenia w algorytmie programowania dynamicznego, możemy więc przyjąć, że rzeczy są ponumerowane tak, jak w tabeli 11.1. Z kolei indeks j przebiega od 1, co 1, aż do maksymalnej pojemności plecaka W . Zatem tablica P dla naszego przykładu ma wymiar 6 wierszy na 23 kolumny. Wartość w polu $P_{i,j}$ definiujemy następująco — jest to wartość optymalnego wypełnienia plecaka o pojemności j rzeczami, których numery mieszczą się między 1 a i .

Algorytm programowania dynamicznego jest sposobem wypełniania pól tablicy P . Pola te są wypełniane wierszami. Zatem wypełniając pierwszy wiersz, dysponujemy tylko rzeczami pierwszego rodzaju, które ważą 6 jednostek każda i mają dla nas wartość 6. Wynika stąd, że dla $i = 1$ oraz dla początkowych pojemności $j = 1, 2, 3, 4, 5$ w odpowiednich polach tablicy znajdują się zera. Dopiero gdy $j = 6$, można w nim umieścić pierwszą sztukę rzeczy nr 1. Następne pozycje aż do $j = 11$ są równe 6, gdyż tylko jedna sztuka pierwszej rzeczy może być włożona do plecaka. Gdy $j = 12$, możemy umieścić już dwie sztuki, a więc w polu o współrzędnych (1, 12) znajdzie się liczba 12, jako wartość dwóch sztuk pierwszej rzeczy. I tak dalej. Kolejna zmiana wartości nastąpi na pozycji (1, 18) i do końca tego wiersza nie ulegnie już zmianie (zobacz tabela 11.2).

Wypełnianie następnych wierszy tablicy P jest już bardziej złożone, gdyż dysponujemy większą ilością różnych rzeczy. Możemy jednak przy tym

korzystać z już wypełnionych wierszy. Reguła korzystania z wcześniej obliczonych wartości w polach tablicy P jest odbiciem tak zwanej: **zasady optymalności Bellmana**, w której poleca się, by:

na każdym kroku podejmować najlepszą decyzję z uwzględnieniem stanu wynikającego z poprzednich decyzji.

W tym szczególnym przypadku, w drugim wierszu, czyli dla $i = 2$, zgodnie z tą zasadą za najlepsze upakowanie plecaka o pojemności j wybieramy albo jego upakowanie rzeczami nr 1 (to znaczy z poprzedniego wiersza tablicy P , czyli dla $i = 1$), albo dokładamy jedną rzecz nr 2 (gdy się zmieści), a resztę pojemności plecaka wypełniamy optymalnie rzeczami nr 1 lub 2. Bardziej konkretnie, jeśli pojemność plecaka j jest taka, że zmieści się w nim rzecz nr 2, czyli $j \geq w_2$, to

wybieramy większą z dwóch wielkości: $P_{1,j}$ i $P_{2,j-w_2} + p_2$ — pierwsza jest

wartością najlepszego wypełnienia plecaka o pojemności j rzeczami nr 1, a druga jest wartością wypełnienia plecaka, składającego się z jednej sztuki rzeczy nr 2 oraz z optymalnego wypełnienia reszty przestrzeni w plecaku, czyli $j - w_2$, rzeczami nr 1 lub 2. Po wypełnieniu drugiego wiersza zgodnie z tą zasadą okazuje się, że najwartościowsze upakowanie plecaka rzeczami nr 1 lub 2 otrzymujemy, zabierając tylko rzeczy nr 2. To było łatwe do przewidzenia na podstawie wartości współczynników: rzecz nr 1 waży 6 jednostek i ma wartość 6, podczas gdy rzecz nr 2 waży 2 jednostki i ma wartość 4, czyli trzy sztuki rzeczy nr 2 mają taką samą wagę, jak jedna rzecz nr 1, ale są dwa razy więcej warte. Wskazują na to wartości w tablicy P , w wierszach 1. i 2.

Dalsze wiersze wypełniamy podobnie. W trzecim wierszu mamy ciekawą sytuację: jeśli pojemność plecaka ma wartość parzystą, to wypełniamy go tylko rzeczami nr 2, a jeśli nieparzystą — to jedną rzecz nr 2 zastępujemy jedną rzeczą nr 3, gdyż jest cenniejsza o jedną jednostkę. Czwarty wiersz jest zdominowany przez wybór rzeczy nr 4, która (jak pamiętamy) ma najlepszy stosunek wartości do wagi. Uwzględnienie rzeczy nr 5 powoduje podobne zmiany jak w trzecim wierszu — dla nieparzystych wartości pojemności plecaka otrzymujemy wartościowsze upakowania, niż gdy dysponujemy tylko czterema pierwszymi rzeczami. Natomiast uwzględnienie rzeczy nr 6 wprowadza zmianę jedynie na pierwszej pozycji w ostatnim wierszu tablicy P .

Tablica P zawiera wartości upakowań plecaka dla różnych jego pojemności, od 1 do 23, i rodzajów rzeczy, od nr 1 do nr 6. Chcielibyśmy również wiedzieć, jaki zestaw rzeczy tworzy poszczególne wypełnienia o obliczonych wartościach. Można to osiągnąć, kojarząc z tablicą P tablicę Q o tych samych wymiarach (zobacz tabela 11.3). Wartością $Q_{i,j}$ jest numer rzeczy, która ostatnia trafiła do plecaka o pojemności j , gdy dysponujemy rzeczami o numerach od 1 do i .

Ogólne wzory na wartości w tablicach P i Q są podane w opisie algorytmu

poniżej.

Tabela 11.3. Tablica Q , skojarzona z tablicą P (przedstawioną w tabeli 11.2) numerów rzeczy dokładanych do plecaka dla poszczególnych jego pojemności i rodzajów rzeczy

j i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	0	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3
4	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	0	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5
6	6	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5

W jaki sposób można odczytać rozwiązanie problemu plecakowego z tych dwóch tablic? Dla naszego przykładu podanego w tabeli 11.2 największa wartość plecaka o pojemności 23 załadowanego rzeczami spośród od 1 do 6 jest dana w polu $P_{6,23}$ i wynosi 80 (oznaczyliśmy ją na zielono). Natomiast rzeczy składające się na wypełnienie plecaka o tej wartości znajdujemy w tablicy Q , zaczynając od pola o tych samych indeksach — jest tam liczba 5. Oznacza to, że rzecz nr 5 była ostatnią rzeczą dorzuconą do plecaka. Ponieważ ta rzecz waży 3 jednostki, więc pozostaje do wypełnienia pojemność plecaka równa $23 - 3 = 20$, patrzymy więc na pole $Q_{6,20}$ — znajduje się tam rzecz nr 4, która waży 2 jednostki. Zatem przenosimy się na pole $Q_{6,18}$, w którym jest również rzecz nr 4. Kontynuujemy to odczytywanie numerów rzeczy z tablicy Q , aż do wyczerpania pojemności plecaka. W tabeli 11.3 kolorem zielonym są oznaczone numery rzeczy tworzących rozwiązanie naszego przykładu. Zatem optymalne upakowanie plecaka o pojemności 23 składa się 10 sztuk rzeczy nr 4 i z jednej sztuki rzeczy nr 5.

Zanim podamy szczegółowy opis przedstawionego algorytmu, zauważmy, że rozwiązanie przykładu z tabeli 11.1 można otrzymać, posługując się jedynie wektorami, czyli tablicami jednowymiarowymi, zamiast dwuwymiarowych. Tablice P i Q są jednak przydatne, gdy interesują nas wyniki rozwiązania tego przykładu dla mniejszej liczby różnych rodzajów rzeczy. W tabelach 11.2 i 11.3 wyróżniliśmy pogrubieniem rozwiązanie dla takich samych danych, jak w tabeli 11.1, ale liczba rzeczy została ograniczona do pierwszych 3, a pojemność

plecaka — do 15.

Ćwiczenie 11.11. Korzystając z tablic P i Q (tabela 11.2 i 11.3), podaj rozwiązania, gdy plecak ma pojemność 19 i masz do dyspozycji 3, 4, 5 pierwszych rodzajów rzeczy, które mogą być do niego zapakowane. ■

Algorytm programowania dynamicznego dla ogólnego problemu plecakowego

Dane: n rzeczy, każda w nieograniczonej ilości, ważących w_i i mających wartość p_i ($i = 1, 2, \dots, n$). Ponadto dana jest maksymalna pojemność plecaka, wynosząca W .

Wyniki: Tablica wartości $P_{i,j}$ najlepszych upakowań plecaka o pojemności j rzeczami rodzajów od 1 do i , dla $i = 1, 2, \dots, n$ oraz $j = 1, 2, \dots, W$; skojarzona tablica rodzajów rzeczy $Q_{i,j}$, dołożonych w ostatnim dopakowaniu plecaka.

Krok 1. {Ustalenie wartości początkowych w tablicach P i Q rozszerzonych, dla ujednolicenia obliczeń, o wiersze i kolumny zerowe.} Dla $j = 1, 2, \dots, W$ przypisz $P_{0,j} := 0$, $Q_{0,j} := 0$. Dla $i = 1, 2, \dots, n$ przypisz $P_{i,0} := 0$, $Q_{i,0} := 0$.

Krok 2. Dla kolejnych rzeczy $i = 1, 2, \dots, n$ wykonaj krok 3.

Krok 3. Dla kolejnych pojemności plecaka $j = 1, 2, \dots, W$ wykonaj krok 4.

Krok 4. **Jeśli** $j \geq w_i$ {czyli plecak jest pojemny, by pomieścić rzecz i }

oraz $P_{i-1,j} < P_{i,j-w_i} + p_i$

to $P_{i,j} := P_{i,j-w_i} + p_i$; $Q_{i,j} := i$

w przeciwnym razie $P_{i,j} := P_{i-1,j}$; $Q_{i,j} := Q_{i-1,j}$ {pozostawiamy wartości z poprzedniego wiersza} ■

Zauważmy, że wzory w kroku 4. na wartości w tablicy P mają postać zależności rekurencyjnych, dla których warunki początkowe są określone w kroku 1.

Powyższy algorytm, wraz z innymi algorytmami z tego rozdziału, dotyczącymi problemu plecakowego, jest zrealizowany w języku Pascal w pakiecie Plecaki, dostępnym na stronie podręcznika. Poniżej zamieszczamy opis tego algorytmu w postaci funkcji w języku Python. Zauważ, w jaki sposób w tej funkcji są definiowane tablice dwuwymiarowe P i Q oraz w jaki sposób odwołujemy się do elementów tych tablic.



```
def ProgDynamPlecakOG(p,w,Waga):
```

```
    n = len(p)
```

```

W = Waga + 1
P = [[0 for col in range(W)] for row in range(n)]
Q = [[0 for col in range(W)] for row in range(n)]
for j in range(1,W):
    P[0][j],Q[0][j] = 0,0
for i in range(1,n):
    P[i][0],Q[i][0] = 0,0
for i in range(1,n):
    for j in range(1,W):
        if j >= w[i] and P[i-1][j] < P[i][j-w[i]] + p[i]:
            P[i][j],Q[i][j] = P[i][j-w[i]]+p[i],i
        else:
            P[i][j],Q[i][j] = P[i-1][j],Q[i-1][j]
return(P,Q)

```

Ćwiczenie 11.12. Utwórz funkcję w języku Python, która na podstawie wyników funkcji ProgDynamPlecakOG tworzy rozwiązanie problemu dla ustalonej liczby rzeczy do wyboru i pojemności plecaka.

Wskazówka. Możesz podpatrzeć rozwiązanie zapisane w języku Pascal w module Plecaki. ■

11.2.3. Algorytm zachłanny dla decyzyjnego problemu plecakowego

W decyzyjnym problemie plecakowym każda rzecz może być wybrana co najwyżej jeden raz. Odpowiada to bardziej rzeczywistym sytuacjom pakowania plecaka. Zmodyfikujemy teraz algorytmy podane dla ogólnego problemu plecakowego tak, aby rozwiązywały ten szczególny przypadek. Jak się okaże, charakteryzują je własności, których nie mają algorytmy rozwiązywania ogólnego problemu.

Zachłanny algorytm dla decyzyjnej wersji problemu plecakowego działa podobnie, jak dla ogólnej wersji. Rzeczy są dobierane do plecaka w kolejności względnych wartości, czyli zgodnie z nierównościami (11.4), a różnica polega jedynie na tym, że możemy wybrać co najwyżej jedną rzecz każdego rodzaju — dzięki temu opis algorytmu jest nieco prostszy.

Algorytm zachłanny dla decyzyjnego problemu plecakowego

Dane: n rzeczy, ważących w_i i mających wartość p_i ($i = 1, 2, \dots, n$); rzeczy zostały ponumerowane tak, że są spełnione nierówności (11.4). Ponadto dana jest maksymalna pojemność plecaka W .

Wyniki: Ilości q_1, q_2, \dots, q_n poszczególnych rzeczy (mogą mieć wartość zero lub jeden), których całkowita waga nie przekracza W .

Krok 1. Dla kolejnych rzeczy $i = 1, 2, \dots, n$, uporządkowanych zgodnie z nierównościami (11.4), wykonaj krok 2.

Krok 2. Jeśli $w_i \leq W$, to przyjmij $q_i = 1$ i przypisz $W := W - w_i$, a w przeciwnym razie przyjmij $q_i := 0$.

Krok 3. Utworzony ładunek plecaka ma wartość $p_1 q_1 + p_2 q_2 + \dots + p_n q_n$. ■

Za pomocą tego algorytmu znajdziemy rozwiązanie dla decyzyjnego problemu plecakowego, który ma takie same wartości współczynników, jak podane w tabeli 11.1 — jedynie wartość W jest mniejsza i wynosi 10. Zgodnie z nierównościami (11.4) ustawiamy rzeczy w kolejności (4, 5, 2, 6, 3, 1) i w takim porządku dobieramy je do plecaka. Wybieramy więc kolejno rzeczy nr: 4, 5, 2 i 6. W sumie ważą one 8 jednostek i mają wartość 23. Nie jest to jednak najwartościowsze upakowanie plecaka. Lepsze otrzymujemy, wstrzymując się z wybraniem rzeczy nr 6 i biorąc rzecz następną w kolejności — nr 3. Wtedy cały plecak jest wypełniony, a jego wartość wynosi 26.

Jeśli przyjmiemy, że waga zawartości plecaka nie może przekroczyć 6 jednostek, to zachłanne rozwiązanie składa się z rzeczy nr: 4, 5 i 6. Widać stąd, że w rozwiązaniu zachłannym niekoniecznie muszą być wybrane kolejne rzeczy z uporządkowania wyznaczonego nierównościami (11.4).

W problemie 11.4 jest naszkicowana zmodyfikowana wersja powyższego algorytmu, która w praktyce daje dobre rozwiązania przybliżone.

Przy niewielkiej modyfikacji decyzyjnego problemu plecakowego algorytm zachłanny może dawać rozwiązania optymalne. Przyjmijmy najpierw osłabione założenie, dotyczące brania rzeczy do plecaka. Brzmi ono: wszystkie rzeczy, z wyjątkiem być może jednej, muszą być w całości, a z jednej rzeczy (nie precyzujemy z której) możemy wziąć dowolną jej część. Odpowiednio do tego założenia modyfikujemy algorytm zachłanny w kroku 2. — jeśli kolejna rzecz nie mieści się w pozostałej części plecaka, to bierzemy z niej tyle, ile zostało wolnego miejsca. W naszym przykładzie dla $W = 10$ rozwiązanie znalezione powyżej uzupełniamy więc o 2/3 rzeczy nr 3 i otrzymujemy upakowany plecak o wartości 26. Dla $W = 6$ z kolei otrzymujemy rozwiązanie: całe rzeczy nr 4 i 5 oraz połowę rzeczy nr 2. Zatem podczas rozwiązywania tego zmodyfikowanego problemu postępujemy tak, jakbyśmy — pakując kolejną rzecz do plecaka — gdy już nie można zmieścić jej w nim w całości, dzielili ją na kawałki i brali tylko tyle z niej, ile miejsca pozostało w plecaku. Zauważmy, że według tak zmodyfikowanego algorytmu zawsze w całości wypełnimy plecak kolejnymi (w uporządkowaniu) rzeczami. Ciekawą własnością rozwiązań generowanych przez ten algorytm dla tak zmodyfikowanego problemu jest to, że otrzymane rozwiązanie jest optymalne! Pamiętajmy o tym podczas pakowania plecaka

przed niedzielną wycieczką w góry.

11.2.4. Programowanie dynamiczne dla decyzyjnego problemu plecakowego

Zmodyfikujemy teraz algorytm programowania dynamicznego tak, aby rozwiązywał decyzyjny problem plecakowy.

Zasadnicza różnica między algorytmami programowania dynamicznego dla obu wersji problemu polega na tym, że w decyzyjnym problemie plecakowym, rozważając rzecz nr i , albo umieszczamy ją w plecaku, albo nie. Zatem w kroku 4. w algorytmie poniżej zysk p_i wynikający z dołożenia tej rzeczy do plecaka dodajemy do wartości rozwiązania dla rzeczy o numerze $i - 1$ oraz wagi plecaka $j - w_i$, czyli do $P_{i-1, j-w_i}$, podczas gdy w algorytmie dla ogólnego problemu ten zysk jest dodawany do wartości rozwiązania, które może zawierać już rzecz o numerze i , czyli do $P_{i, j-w_i}$.

Ponieważ w iteracji i możemy wybrać tylko jedną rzecz o numerze i , więc w tablicy Q , skojarzonej z tablicą P , ten wybór oznaczamy cyfrą 1 zamiast numeru rzeczy, jak to robiliśmy w przypadku ogólnego problemu plecakowego.

Algorytm programowania dynamicznego dla decyzyjnego problemu plecakowego

Dane: n rzeczy, ważących w_i i mających wartość p_i ($i = 1, 2, \dots, n$); rzeczy zostały ponumerowane tak, że spełnione są nierówności (11.4). Ponadto dana jest maksymalna pojemność plecaka W .

Wyniki: Tablica wartości $P_{i,j}$ najlepszych upakowań plecaka o pojemności j rzeczami rodzajów od 1 do i , dla $i = 1, 2, \dots, n$ oraz $j = 1, 2, \dots, W$; skojarzona tablica $Q_{i,j}$ zawierająca informacje o wybranych rzeczach.

Krok 1. {Ustalenie wartości początkowych w tablicach P i Q rozszerzonych, dla ujednolicenia obliczeń, o wiersze i kolumny zerowe.} Dla $j = 1, 2, \dots, W$ przypisz $P_{0,j} := 0$, $Q_{0,j} := 0$. Dla $i = 1, 2, \dots, n$ przypisz $P_{i,0} := 0$, $Q_{i,0} := 0$.

Krok 2. Dla kolejnych rzeczy $i = 1, 2, \dots, n$ wykonaj krok 3.

Krok 3. Dla kolejnych pojemności plecaka $j = 1, 2, \dots, W$ wykonaj krok 4.

Krok 4. **Jeśli** $j \geq w_i$ {Czyli plecak jest pojemny, by pomieścić rzecz nr i .}

oraz $P_{i-1,j} < P_{i-1,j-w_i} + p_i$

to $P_{i,j} := P_{i-1,j-w_i} + p_i$; $Q_{i,j} := 1$

w przeciwnym razie $P_{i,j} := P_{i-1,j}$; $Q_{i,j} := 0$. ■

W tabelach 11.4 i 11.5 są zamieszczone wyniki działania tego algorytmu dla danych z tabeli 11.1 oraz dla $W = 10$. Zauważmy na tym przykładzie, jaka jest różnica między rozwiązaniami obu wersji problemu plecakowego. Wyniki rozwiązania, czyli wartość plecaka, odczytujemy w taki sam sposób, wybierając w tablicy P element, który odpowiada liczbie rzeczy (indeks wiersza) i dopuszczalnej pojemności plecaka (indeks kolumny). Dla naszego przykładu, w którym $n = 6$ i $W = 10$, otrzymujemy $P_{6,10} = 26$. Z kolei rzeczy zapakowane do plecaka odczytujemy z tablicy Q . W odróżnieniu od ogólnego problemu plecakowego, musimy posłużyć się całą tablicą Q , a nie tylko jej ostatnim wierszem. Wartość $Q_{6,10} = 0$ oznacza, że rzecz nr 6 nie należy do rozwiązania, sprawdzamy więc element $Q_{5,10}$ — jego wartość wynosi 1, zatem rzecz nr 5 należy do rozwiązania. Rzecz nr 5 waży 3 jednostki, więc w następnym kroku sprawdzamy element $Q_{4,10-3}$ — jego wartość równa 1 oznacza, że rzecz nr 4 również należy do rozwiązania. Kontynuujemy to postępowanie w następnych wierszach 3, 2, 1 i ostatecznie otrzymujemy rozwiązanie, które składa się z rzeczy 2, 3, 4 i 5. Rozwiązanie to jest oznaczone w tabeli 11.4 i 11.5 zielonym kolorem, a liczby, które stanowią rozwiązanie dla $n = 4$ i $W = 10$ — pogrubieniem.

Tabela 11.4. Tablica P wartości upakowań plecaka, wygenerowanych przez algorytm programowania dynamicznego zastosowany do rozwiązania przykładu decyzyjnego problemu plecakowego podanego w tabeli 11.1

$\begin{matrix} j \\ i \end{matrix}$	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	6	6	6	6	6
2	0	4	4	4	4	6	6	10	10	10
3	0	4	5	5	9	9	9	10	11	11
4	0	7	7	11	12	12	16	16	16	17
5	0	7	10	11	17	17	21	22	22	26
6	2	7	10	12	17	19	21	23	24	26

Tabela 11.5. Tablica Q , skojarzona z tablicą P (przedstawioną w tabeli 11.4) —

cyfra 1 na pozycji (i, j) oznacza wzięcie i -tej rzeczy do plecaka o pojemności j

j i	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	1	1	1	1	1
2	0	1	1	1	1	0	0	1	1	1
3	0	0	1	1	1	1	1	0	1	1
4	0	1	1	1	1	1	1	1	1	1
5	0	0	1	0	1	1	1	1	1	1
6	1	0	0	1	0	1	0	1	1	0

Te przykłady wskazują, że nie można uprościć algorytmu programowania dynamicznego dla decyzyjnego problemu plecakowego tak, jak to sugerujemy dla ogólnego problemu plecakowego w zadaniu 11.3.

Ćwiczenie 11.13. Opisz w języku Python algorytm programowania dynamicznego dla decyzyjnego problemu plecakowego. Zmodyfikuj w tym celu funkcję ProgDynamPlecakOG. Następnie wykonaj obliczenia dla powyższych przykładów. Napisz także funkcję, odpowiadającą funkcji z ćwiczenia 11.12, dla tej wersji problemu plecakowego. ■

11.2.5. Złożoność algorytmów rozwiązywania problemów plecakowych

W obu algorytmach zachłannych jest wykonywana podobna liczba operacji: najpierw musimy uporządkować rzeczy zgodnie z nierównościami (11.4), a potem przeglądamy je po kolei. Zatem złożoność tych algorytmów jest zdominowana przez złożoność porządkowania, która dla najszybszego algorytmu jest proporcjonalna do $n \log_2 n$ (zobacz punkt 10.2.2 i 10.4).

Oba algorytmy programowania dynamicznego mają również bardzo podobną złożoność obliczeniową. Każdy z tych algorytmów jest podwójną iteracją: pierwsza przebiega po kolejnych rzeczach, czyli od 1 do n , a druga, głębsza — po kolejnych dopuszczalnych pojemnościach plecaka, czyli od 1 do maksymalnej pojemności, wynoszącej W . Odbiciem takiego działania tych algorytmów są wyniki, które tworzą prostokątne tablice o wymiarach n na W . Wypełnienie każdej pozycji w tych tablicach jest związane z wykonaniem kilku operacji: dodawania, porównania, przypisania. Zatem całkowita złożoność algorytmów

programowania dynamicznego jest proporcjonalna do liczby elementów w tych tablicach, czyli do nW .

Zwróćmy w tym miejscu uwagę, że ta złożoność algorytmów programowania dynamicznego zależy od liczby n rzeczy i od maksymalnej pojemności W plecaka, zależy więc od **liczby** danych i od **wartości** jednej z danych. Taka złożoność jest charakterystyczna dla algorytmów wykorzystujących zasadę programowania dynamicznego. W praktyce algorytmy o takiej złożoności są efektywne pod warunkiem, że wartości pojemności W nie są zbyt duże. Jeśli problem plecakowy dotyczy rzeczywiście plecaka, to wartość W jest na ogół niewielka, może wynieść 20 kg, ale nie więcej niż 50 kg. Ale, jak nadmieniliśmy, ten problem może się również odnosić do ładowania dużych samochodów, samolotów i statków, których pojemność W jest duża. Wówczas algorytmy programowania dynamicznego stają się mało praktyczne.

11.3. Zadania i problemy

Problem 11.1. Zaproponuj algorytm rozwiązania zadania, jakie miał wykonać Tezeusz. Co będzie grało w nim rolę nici Ariadny? Sprawdź poprawność działania swojego algorytmu na przykładzie labiryntu przedstawionego na rysunku 11.1.

Zadanie 11.2. Jaki wpływ na działanie algorytmu znajdowania wyjścia z labiryntu będzie miała zmiana kolejności kierunków poruszania się, np. na (G, P, L, D)? Porównaj działanie tak zmienionego algorytmu z oryginalnym algorytmem, dla pól $s = 4a$, $3a$ i $2a$ w labiryncie pokazanym na rysunku 11.1.

Problem 11.2. Zapoznaj się z opisami algorytmów przeszukiwania z nawrotami (lub w głąb) i znajdowania najkrótszej drogi w grafie, znajdującymi się w plikach `wyjscie.pas` i `dijkstra.pas`, dostępnych na stronie towarzyszącej książce, i przygotuj dla nich dane, które modelują labirynt przedstawiony na rysunku 11.1, a następnie wykonaj obliczenia dla przykładów omówionych w tekście.

Zadanie 11.3. Jak wspomnieliśmy, opis algorytmu programowania dynamicznego dla ogólnego problemu plecakowego można uprościć, wykorzystując tablice jednowymiarowe zamiast tablic P i Q . Wtedy oczywiście będziemy mogli znaleźć rozwiązania dla różnych pojemności plecaka, mając do dyspozycji wszystkie rodzaje rzeczy. Podaj opis tej uproszczonej wersji algorytmu, uprość również odpowiednio opisy komputerowej realizacji tego algorytmu.

Zadanie 11.4. Jedną z najczęściej wykonywanych operacji kasowych jest wydawanie reszty. Spotykamy się przy tym z różnymi metodami i zwyczajami kasjerek i kasjerów. Najbardziej denerwująca jest chęć pozbycia się przez nich największej ilości drobnych. Z kolei z szacunku do klienta powinno wynikać, że otrzymujemy najmniejszą liczbę różnych banknotów i monet. **Problem reszty**^[2]

właśnie na tym polega: dla danej kwoty M pieniędzy oraz nominałów banknotów i monet $c_k, c_{k-1}, \dots, c_2, c_1$, spełniających nierówności $c_k > c_{k-1} > \dots > c_2 > c_1 = 1$, należy określić najmniejszą liczbę banknotów i monet, którymi kwota M może być wydana jako reszta. Dla wygody kasjerek i kasjerów byłoby dobrze, gdyby istniała prosta metoda znajdowania takiej postaci reszty, aby za każdym razem nie musieli oni rozwiązywać trudnego problemu matematycznego.

Spotykamy się ze zwyczajem, że reszta M jest wydawana w następujący sposób: w kolejności od największych nominałów, pozostałą do wydania kwotę otrzymujemy w największej liczbie banknotów lub monet, największego mieszczącego się w niej nominału. A więc, jeśli reszta wynosi 96 groszy, to otrzymamy ją jako sumę groszy: $50 + 2 \cdot 20 + 5 + 1$. Jest to zapewne wygodniejsza do przechowania i noszenia liczba monet niż 96 jednogroszówek. Opisany sposób wydawania reszty można nazwać zachłannym, gdyż na każdym kroku jest minimalizowana liczba wszystkich banknotów i monet przez składanie reszty z największej liczby banknotów lub monet największego mieszczącego się w niej nominału.

(A) Podaj ścisły opis naszkicowanego wyżej zachłannego algorytmu wydawania reszty M możliwie najmniejszą liczbą nominałów $c_k, c_{k-1}, \dots, c_2, c_1$. Utwórz realizację tego algorytmu w arkuszu kalkulacyjnym i sporządź odpowiedni opis w języku Pascal lub Python.

(B) Posłuż się reprezentacją algorytmu utworzoną w punkcie (A) i utwórz resztę dla kwot: 91 gr, 7 zł i 39 gr, 10 zł i 41 gr.

Problem 11.3. Podaj przykład zbioru nominałów monet fikcyjnej waluty *prime*, dla którego zachłanny algorytm wydawania reszty opisany w zadaniu 11.4 nie zawsze „wydaje” resztę w postaci najmniejszej liczby monet.

Problem 11.4. Algorytm zachłanny dla decyzyjnego problemu plecakowego ma następujące uogólnienie: wykonaj n iteracji, w iteracji k najpierw umieść w plecaku rzecz nr k i dalej wypełniaj plecak zgodnie z regułą zachłanną, pomijając rzecz umieszczoną już w plecaku na początku. Rzeczy rozpatruj w kolejności wyznaczonej na podstawie nierówności (11.4). Jako rozwiązanie problemu wybierz najlepsze, jakie otrzymasz w n iteracjach.

Podaj ścisły opis tak zmodyfikowanego algorytmu. Zastosuj go do rozwiązania przykładów omówionych w tekście. Podaj przykład problemu, dla którego i ten algorytm nie generuje rozwiązania optymalnego.

Problem 11.5. **Problem podziału.** Dysponujesz n rzeczami o wagach w_1, w_2, \dots , w jednostkach każda. Opracuj algorytm programowania dynamicznego, który będzie sprawdzał, czy te rzeczy można rozłożyć na dwa tyle samo ważące plecaki. Dla uproszczenia możesz założyć, że suma wag wszystkich rzeczy jest liczbą parzystą. Zapisz swój algorytm w języku Pascal lub Python, a następnie sprawdź jego działanie na danych, złożonych z wag dziewięciu rzeczy: 3, 4, 3, 1,

3, 2, 3, 2, 1, których sumaryczna waga wynosi 22.

Wskazówka. Twój algorytm powinien wypełniać tablicę o wymiarach n na V , gdzie V jest połową sumy wszystkich wag, o elementach 0 lub 1. Element (i, j) w tej tablicy ma przyjmować wartość 1, gdy wśród i pierwszych rzeczy można wybrać podzbiór ważący w sumie j , a wartość 0 — w przeciwnym razie.

Mogłeś zapoznać się z dwiema nowymi metodami rozwiązywania problemów, które wykorzystują:

- ▶ **decyzje zachłanne;**

- ▶ **programowanie dynamiczne;**

oraz poznać ich działanie w algorytmach znajdowania:

- ▶ **drogi do wyjścia z labiryntu;**

- ▶ **najkrótszej drogi do wyjścia;**

- ▶ **zachłannego upakowania plecaka;**

- ▶ **najcenniejszego upakowanie plecaka.**

[\[1\]](#) Problemowi wyjścia z labiryntu są poświęcone punkty 12.2 i 14.2 w książce [Piramidy].

[\[2\]](#) Problemowi reszty jest poświęcony punkt 12.1 w książce [Piramidy].

Rozdział 12. Własności algorytmów — podsumowanie

Masz tutaj okazję spojrzeć na poznane algorytmy i:

- ▶ podsumować swoją wiedzę o ich **ogólnych własnościach**;
- ▶ ponownie zastanowić się, czy są one **poprawne i efektywne**.

Pierwszy krok w tym podsumowaniu już wykonaliśmy w punkcie 10.4, pisząc o ogólnych własnościach algorytmów porządkowania, które stanowią w tej książce największą grupę metod rozwiązywania jednego problemu.

12.1. Algorytmy — spojrzenie z lotu ptaka

Jeśli nawet poznałeś tylko część algorytmów z tej książki wraz z metodami ich tworzenia, to już możesz pokusić się o określenie własności i sformułowanie ogólnych cech, jakie powinien mieć każdy algorytm. Od samego początku te cechy algorytmów są przez nas uwzględniane, chociaż na ogół nie piszemy o tym wprost. Dysponując teraz wieloma przykładami algorytmów, jest nam łatwiej wymienić te cechy oraz uzasadnić, że rzeczywiście są one niezbędne, by algorytm prowadził do rozwiązania postawionego problemu.

Najważniejsze własności algorytmów można podzielić na trzy grupy:

1. Poprawność.
2. Skończoność.
3. Efektywność.

Algorytm poprawny rzeczywiście rozwiązuje problem, dla którego został opracowany. Poprawność algorytmu określamy względem specyfikacji problemu. Każdy algorytm jest opisany w tej książce w postaci listy kroków, którą poprzedza specyfikacja problemu i względem tej specyfikacji możemy określać wszystkie jego własności, w tym — czy algorytm jest rzeczywiście poprawny. Przypomnijmy więc sobie, co to jest specyfikacja (wprowadziliśmy to pojęcie w punkcie 1.3.1) i powiedzmy, kiedy algorytm jest poprawny.

Specyfikacja jest ścisłą definicją problemu i składa się z **danych, wyników** oraz opisu **związków zachodzących między nimi**. Zatem specyfikacja to coś więcej niż tylko dane i wyniki, gdyż podane w niej są: dane i warunek, jaki muszą one spełniać, zwany **warunkiem początkowym**, oraz wyniki i warunek, zwany **warunkiem końcowym**, jaki muszą one spełniać. Warunek końcowy określa również związek wyników z danymi.

Posłużymy się przykładem problemu przeszukiwania zbioru uporządkowanego. W algorytmie binarnego przeszukiwania (zobacz punkt 9.2) podaliśmy następującą specyfikację tego problemu:

Problem przeszukiwania zbioru uporządkowanego

Dane: Uporządkowany ciąg liczb w tablicy $a[k..l]$, gdzie $k \leq l$, to znaczy $a_k \leq a_{k+1} \leq \dots \leq a_l$; oraz element y spełniający nierówności $a_k \leq y \leq a_l$.

Wynik: Takie s ($k \leq s \leq l$), że $a_s = y$, lub przyjąć $s = -1$, gdy $y \neq a_i$ dla każdego i ($k \leq i \leq l$). ■

Danymi w tym problemie są: ciąg liczb zapisanych w tablicy a oraz dodatkowa liczba y . Warunkiem początkowym jest, że liczby w tablicy a są uporządkowane, a dodatkowa liczba y nie jest mniejsza od pierwszej liczby w tablicy i nie jest większa od ostatniej. Wynikiem natomiast jest liczba naturalna s , która jest indeksem elementu w tablicy, równego y , lub wynosi -1 , gdy takiego elementu nie ma — i to jest warunek końcowy nałożony na wynik, będący jednocześnie warunkiem wiążącym wynik z danymi. W punkcie 12.2 posłużymy się tą specyfikacją, by uzasadnić ściśle, że algorytm binarnego przeszukiwania rzeczywiście rozwiązuje ten problem. Zdefiniujmy więc najpierw ściśle poprawność algorytmu.

Algorytm jest poprawny, jeżeli działa zgodnie ze specyfikacją, to znaczy dla każdego poprawnych danych daje odpowiedni wynik. Bardziej formalnie mówimy, że algorytm jest **poprawny** (a dokładniej, **całkowicie poprawny**) względem warunku początkowego i warunku końcowego, gdy dla każdego danych spełniających warunek początkowy obliczenia algorytmu kończą się i wyniki spełniają warunek końcowy. Rozważa się również częściową poprawność algorytmów. Algorytm nazywa się **częściowo poprawny**, jeśli dla każdego obliczeń, które się kończą, wynik jest poprawny względem warunku początkowego i końcowego. Zatem, weryfikując częściową poprawność, nie musimy wykazać, że obliczenia kończą się dla wszystkich poprawnych danych. Jak można sprawdzić, czy konkretny algorytm jest całkowicie lub częściowo poprawny? Przecież ma to zachodzić dla wszystkich poprawnych danych, których może być nieskończenie wiele.

W pojęciu poprawności algorytmu kryją się jeszcze dwie inne cechy, o których nie mówimy w tej książce wprost, ale o których pamiętaliśmy, wyprowadzając każdy z algorytmów: dobra określoność opisu oraz ogólność, zwana również uniwersalnością algorytmu.

Dobra określoność opisu algorytmu oznacza, że wszystkie występujące w opisie polecenia i operacje są zrozumiałe dla czytającego i mogą być przez niego wykonane, oraz kolejność działań w algorytmie nie pozostawia żadnej dowolności co do porządku ich wykonywania. Konsekwencją i sprawdzianem dobrej określoności opisu algorytmu może być możliwość jednoznacznego

reprezentowania go w postaci jeszcze bardziej sformalizowanej, np. w programie zapisanym w języku Pascal lub Python.

Ogólność czy **uniwersalność** algorytmu jest jego cechą ściśle związaną z rozwiązywanym problemem: algorytm ma służyć do rozwiązywania klasy zadań, które spełniają specyfikację problemu, a mogą różnić się wartościami danych. Na przykład, jeśli algorytm jest przeznaczony do rozwiązywania równania kwadratowego, to można go zastosować do rozwiązywania równania o dowolnych współczynnikach, jedynie współczynnik przy kwadracie zmiennej powinien być różny od zera, by rzeczywiście było to równanie kwadratowe (zobacz punkt 3.1). Podobnie, z każdego algorytmu porządkowania n liczb można korzystać, chcąc znaleźć porządek wśród dowolnych n liczb, nawet jeśli $n = 1$ lub, gdy te liczby są już uporządkowane (zobacz punkt 10.4).

Chociaż **skończoność** algorytmu jest jedną z cech wymaganych do jego poprawności, wymieniamy tę własność osobno, gdyż można ją rozważać niezależnie od formalnej poprawności.

Nawet jeśli algorytm charakteryzuje się skończonością działania, to może nie być praktyczną metodą rozwiązywania problemu dla dużej liczby danych (zobacz np. tabelę 1.1). I nie ma wówczas znaczenia, czy posługujemy się w tym celu komputerem ani jak szybkim. W ostatnich latach znacznie wzrosła moc komputerów, ale nastąpił jeszcze większy wzrost rozmiarów problemów, które człowiek chciałby rozwiązywać (zobacz Uwagi do problemu 13.35). Dlatego nieodłączną cechą dobrego algorytmu stała się jego duża **efektywność**, czyli mała złożoność obliczeniowa, gwarantująca jego użyteczność w praktycznych sytuacjach.

12.2. Poprawność algorytmów

Poprawność algorytmu jest definiowana w algorytmice dość ściśle. Wiedza matematyczna zdobywana w szkole jest jednak niewystarczająca, byśmy mogli prowadzić w tej książce takie rozważania w pełni rygorystycznie. Wystarczy powiedzieć, że dowód poprawności algorytmu ma cechy dowodu twierdzenia matematycznego. Trzeba bowiem wykazać, że pewne stwierdzenia zachodzą dla, na ogół, nieskończonej liczby możliwych danych poprawnych i że ciąg działań jest skończony. Poprawność algorytmów jest jednak bardzo ważną ich cechą, zadbaliliśmy więc w tej książce, by wszystkie podane przez nas algorytmy były poprawne, chociaż tego nie dowodzimy formalnie. Jednak każdy algorytm szczegółowo uzasadniamy w trakcie jego budowania.

Dla przykładu, podamy formalny dowód całkowitej poprawności algorytmu binarnego przeszukiwania (zobacz punkt 9.2), który przytoczymy ponownie wraz z pełną specyfikacją.

Algorytm binarnego przeszukiwania

Dane: Uporządkowany ciąg liczb w tablicy $a[k..l]$, gdzie $k \leq l$, tzn. $a_k \leq a_{k+1} \leq \dots \leq a_l$; oraz element y spełniający nierówności $a_k \leq y \leq a_l$.

Wynik: Takie s ($k \leq s \leq l$), że $a_s = y$, lub przyjąć $s = -1$, gdy $y \neq a_i$ dla każdego i ($k \leq i \leq l$).

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

Krok 2. Jeśli $lewy > prawy$, to przypisz $s := -1$ i zakończ algorytm. {Elementu y nie ma w przeszukiwanej tablicy.}

Krok 3. $s := (lewy + prawy) \text{ div } 2$; Jeśli $a_s = y$, to zakończ algorytm. {Znaleziono element y w przeszukiwanej tablicy.}

Jeśli $a_s < y$, to $lewy := s + 1$, a w przeciwnym razie $prawy := s - 1$.

Wróć do kroku 2. ■

W poprzednim punkcie szczegółowo skomentowaliśmy specyfikację tego problemu. Teraz do udowodnienia całkowitej poprawności tego algorytmu musimy wykazać, że:

1. Dla każdego wykonania algorytmu, które rozpoczyna się z danymi spełniającymi warunek początkowy (opisany w punkcie Dane), jeśli obliczenia dochodzą do końca, to wynik, czyli wartość s , spełnia warunek końcowy (opisany w punkcie Wynik).

2. Dla każdych danych spełniających warunek początkowy obliczenia w algorytmie się kończą.

Aby to wykazać, należy oczywiście skorzystać z tego, jakie działania są wykonywane w algorytmie między warunkiem początkowym (czyli początkiem algorytmu) a warunkiem końcowym (czyli zakończeniem algorytmu). Do udowodnienia poprawności działania tego algorytmu posłużymy się następującym warunkiem, który jest tak zwanym **niezmiennikiem iteracji** (pętli) wykonywanej w krokach 2. i 3.:

jeśli istnieje s takie, że $lewy \leq s \leq prawy$ i $a_s = y$, to $a_{lewy} \leq y \leq a_{prawy}$ (12.1)

Ten warunek oznacza, że jeśli element y znajduje się w przeszukiwanej tablicy, to w trakcie działania algorytmu znajduje się w tej części tablicy, która jest jeszcze przeszukiwana.

Dowód całkowitej poprawności algorytmu binarnego przeszukiwania polega teraz na sprawdzeniu, czy są spełnione następujące trzy warunki:

1. Niezmiennik (12.1) jest prawdziwy przed wykonaniem pierwszej iteracji

kroków 2. i 3.

2. Jeśli niezmiennik (12.1) jest prawdziwy przed wykonaniem iteracji kroków 2. i 3., to po zakończeniu iteracji pozostaje również prawdziwy.

3. Algorytm kończy działanie, czyli kończy się iteracja kroków 2. i 3., a wynik spełnia warunek końcowy.

Własność 1. wynika wprost z warunku początkowego, jaki spełniają dane, gdyż założyliśmy, że $ak \leq y \leq al$, a w kroku 1. przyjmujemy, że $lewy := k$; $prawy := l$.

Do wykazania spełnienia warunku 2. założmy, że przed kolejną iteracją kroków 2. i 3. jest spełniony niezmiennik (12.1). Zauważmy teraz, że dla nowej wartości s , obliczonej na początku kroku 3., mamy $lewy \leq s \leq prawy$, gdyż średnia z dwóch liczb nie jest mniejsza od mniejszej z nich. W dalszej części tego kroku albo okazuje się, że $as = y$ — wtedy algorytm kończy działanie, albo nie znaleźliśmy jeszcze takiego s . Wtedy jeśli $as < y$, to y może być równy elementowi z indeksem co najmniej $s + 1$, czyli $as + 1 \leq y \leq aprawy$, a więc możemy przyjąć $lewy := s + 1$. W przeciwnym razie mamy $y < as$, a więc z podobnych względów możemy przyjąć $prawy := s - 1$. Zatem w obu tych przypadkach, gdy nie został jeszcze znaleziony element y w tablicy a , po zakończeniu kolejnej iteracji kroków 2. i 3. niezmiennik (12.1) jest nadal spełniony.

Do udowodnienia całkowitej poprawności algorytmu musimy jeszcze wykazać własność 3. Algorytm może zakończyć działanie na początku kroku 2. lub w kroku 3. Zakończenie w kroku 3. jest oczywiste — znaleziony został w tablicy a element równy y . Poprawność zakończenia algorytmu w kroku 2. wynika z niezmiennika (12.1). Na podstawie tego warunku wiemy bowiem, że jeśli element y znajduje się w tablicy, to $alewy \leq y \leq aprawy$. W kroku 2. natomiast jest spełniony warunek $lewy > prawy$, stąd wynika, że elementu y nie ma w tablicy a . Dodatkowo zauważmy, że iteracja kroków 2. i 3. musi się zakończyć w jednym z tych przypadków, gdyż w danej iteracji kroku 3., jeśli nie znaleziono elementu y w tablicy a , to długość przeszukiwanego przedziału jest zmniejszana przynajmniej o jeden element (albo zwiększamy $lewy$ o jeden, albo zmniejszamy $prawy$ o jeden).

Wykazaliśmy więc, stosując niezmiennik (12.1), że algorytm binarnego przeszukiwania jest całkowicie poprawny. Zastosowaliśmy metodę dowodzenia poprawności programów, która nazywa się **metodą niezmienników**.



Czytelnicy, którzy chcą rozszerzyć swoją wiedzę na temat ścisłego dowodzenia poprawności algorytmów, mogą prześledzić elementarne rozważania na ten temat zawarte w książkach [Harel, rozdział 5.] i [Bentley]. Bardziej

Poprawność algorytmu a poprawność programu

Ponieważ definicje poprawności algorytmu i poprawności działania programu (będącego jego realizacją) są takie same, do sprawdzenia poprawności algorytmu można się posłużyć komputerem. Realizacją algorytmu może być program napisany w wybranym języku programowania. Sprawdzenie poprawności programu, zarówno za pomocą komputera, jak i na podstawie analizy jego tekstu, może być łatwiejsze niż posługiwanie się opisem algorytmu, gdyż instrukcje w programach są bardziej precyzyjne niż słowne opisy algorytmów.

Dla większości algorytmów przedstawionych w tej książce zamieściliśmy ich realizacje w postaci programów w językach Pascal i Python, a komputerowe realizacje wszystkich algorytmów są dostępne na stronie związanej z tą książką: <http://edukacja.helion.pl/algorytmika>. Mamy nadzieję, że będą one wielokrotnie wykorzystywane przez Czytelników do sprawdzenia poprawności działania algorytmów.

Program może zawierać dwa rodzaje błędów: składniowe i logiczne. **Błąd składniowy** jest niezgodnością treści programu z zasadami pisania programów. W usuwaniu błędów składniowych pomaga nam zazwyczaj system, w którym piszemy program. Usunięcie z programu błędów składniowych nie oznacza jednak jeszcze, że nie ma w nim **błędów logicznych**, co oznaczałoby, że program rozwiązuje problem zgodnie ze specyfikacją; dla każdego danych spełniających warunek początkowy daje wynik spełniający warunek końcowy. Błędy logiczne powodują, że nie dla wszystkich poprawnych danych program generuje poprawne wyniki. Takie błędy jest zwykle trudniej znaleźć niż błędy składniowe — nie są one na ogół sygnalizowane przez komputer — otrzymujemy po prostu złe wyniki. Może być wiele powodów złego działania programu, który jest składniowo poprawny. Mogliśmy na przykład źle zrozumieć znaczenie instrukcji systemu (programu), w którym jest pisany program. Poważniejszy rodzaj błędów stanowią błędy w algorytmach, które na ogół są niemal automatycznie przenoszone do programów, będących ich realizacją.

Uruchamianie i testowanie programów

Sprawdzanie, czy program nie zawiera błędów, nazywa się **uruchamianiem programu**. Dosłownie mówi się często — „odpluskwanie” programu (ang. *debugging*), gdyż, jak głosi historia, przyczyną jednego z błędów w działaniu pierwszego komputera był zakleszczony między elektronicznymi elementami insekt (ang. *bug*). Częścią uruchamiania programu jest **testowanie**, czyli wykonywanie go na testowych przykładach danych. Testy służą do sprawdzania, czy program poprawnie przechodzi przez wszystkie „newralgiczne” punkty algorytmu, np. przez rozgałęzienia. W wielu miejscach tej książki analizujemy

działanie algorytmów na takich właśnie danych.

Wykrywanie błędów logicznych w programach jest trudniejszym zadaniem. Najprostsza metoda polega na testowaniu poprawności programu z zastosowaniem specjalnie określonych danych testowych, dla których wiemy, jak powinien on działać i jaki ma być wynik. W testowaniu programu można również wykorzystać instrukcje drukowania wartości pośrednich (umieszczone na czas testowania w programie), które potrafimy zweryfikować. Systemy uruchamiania programów w językach Pascal i Python zawierają wiele udogodnień ułatwiających wszechstronne testowanie programów — z wielu z nich korzystaliśmy, przygotowując programy podane w tej książce.

Przedstawiliśmy wiele algorytmów i programów, które rozwiązują zadania rachunkowe — są one na ogół realizacjami wzorów matematycznych. Sprawdzenie poprawności takiego programu jest dość łatwe, gdyż polega na porównaniu wzoru z jego zapisem w programie. Pojawia się natomiast inny problem — zgodność wyników obliczeń komputerowych z dokładnymi wartościami rozwiązań matematycznych, gdy obliczenia są wykonywane na liczbach reprezentowanych w komputerze w sposób przybliżony. W punktach 3.1 i 3.3 podaliśmy przykłady różnych rodzajów niedokładności, pojawiających się w obliczeniach matematycznych, których powodem są przede wszystkim błędy zaokrągleń w reprezentacjach liczb oraz niestabilność algorytmów.

Na zakończenie tych uwag o poprawności algorytmów i programów podkreślimy z całym przekonaniem, że uruchamianie programu wraz z jego testowaniem nie gwarantuje otrzymania w pełni poprawnego algorytmu czy programu. Techniki te można stosować do wykrywania błędów, ale ich użycie nie wystarcza do wykazania braku błędów w programach. Jedynie dowód formalnej poprawności algorytmu jest godny zaufania.

12.3. Skończoność algorytmów

Badanie i dowodzenie skończoności działania algorytmu powinno być częścią weryfikacji jego poprawności. Wynika to z dość oczywistego wymagania, że algorytm można uznać za poprawny, gdy daje rozwiązanie w skończonej liczbie kroków, czyli — gdy jest skuteczny.

Podobnie jak poprawność algorytmów, również ich skończoność objaśnimy krótko na przykładach, natomiast możemy zapewnić, że dołożyliśmy wszelkich starań, by wszystkie wyprowadzone w tej książce algorytmy miały skończone działanie.

Na skończoność algorytmu mają wpływ przede wszystkim iteracje, zwłaszcza warunkowe instrukcje iteracyjne, które powinny zawierać dobrze określony warunek zakończenia. Za iterację uznajemy tutaj również ciąg wywołań

rekurencyjnych.

W poprzednim punkcie wykazaliśmy już skończoność działania algorytmu binarnego przeszukiwania. W tym celu posłużyliśmy się niezmiennikiem (12.1) oraz spostrzeżeniem, że w każdej iteracji kroków 2. i 3. przedział wyznaczony przez dwa wskaźniki, *lewy* i *prawy*, jest zmniejszany. Zatem albo zostaje znalezione takie s , że $as = y$, albo staje się $lewy > prawy$. W obu przypadkach algorytm kończy swoje działanie.

Powodem niezatrzymywania się algorytmu (programu) może być sprawdzanie spełnienia dokładnych równości wielkości, które w algorytmach mogą przyjmować niedokładne wartości, np. którym w programach odpowiadają zmienne typu rzeczywistego. Swojego działania może nie kończyć również algorytm taki, jak iteracyjne znajdowanie wartości pierwiastka kwadratowego (zobacz punkt 7.7) lub znajdowanie zera funkcji metodą połowienia przedziału (zobacz punkt 9.4), gdy chcemy uzyskać zbyt dużą dokładność wyniku (np. większą niż wynosi dokładność arytmetyki komputerowej, w której są wykonywane obliczenia).

Badanie skończoności działania algorytmu może się odbywać w trakcie uruchamiania i testowania programu, gdy chcemy sprawdzić, czy dla danych testowych program zawsze się zatrzymuje.

Podamy jeszcze trzy przykłady bardzo prostych algorytmów o dość zaskakujących własnościach, związanych z ich skończonością działania.

Znany jest następujący algorytm generowania liczb naturalnych: rozpocznij od danej liczby n i w kolejnych krokach, dopóki $n \neq 1$, wykonuj: jeśli liczba n jest parzysta, to podziel ją na pół, a jeśli jest nieparzysta, to zwiększ ją trzykrotnie i dodaj 1. Na przykład, gdy $n = 3$, otrzymujemy ciąg: 3, 10, 5, 16, 8, 4, 2, 1. Sformułujmy ten algorytm ściśle.

Algorytm generowania ciągu liczb naturalnych Ciąg-3

Dane: Liczba naturalna n .

Wynik: Ciąg liczb naturalnych rozpoczynających się daną liczbą n , którego kolejne elementy są obliczane w kroku 1.

Krok 1. Dopóki $n \neq 1$, wykonuj: jeśli liczba n jest parzysta, to przyjmij $n := n \text{ div } 2$, a w przeciwnym razie przyjmij $n := 3n + 1$. Dopisz do ciągu obliczoną wartość n . ■

Zatem powyższy algorytm zatrzymuje się, gdy $n = 3$. Zatrzymuje się również, gdy $n = 1, 2$ i 4. Chociaż dla niektórych liczb naturalnych generowane ciągi wartości liczby n są dość niezwykłe, to jednak do dzisiaj nie natrafiono na taką wartość liczby n , dla której ciąg generowany przez ten algorytm jest nieskończony! Nikt również nie udowodnił, że ten prosty algorytm ma własność stopu, czyli że zatrzymuje się w każdym przypadku. Możesz jednak wykonać

proste ćwiczenie.

Ćwiczenie 12.1. Wykaż, że algorytm Ciąg-3 kończy swoje działanie wtedy i tylko wtedy, gdy w generowanym ciągu pojawia się liczba, która jest potęgą liczby 2. ■

Rozważmy dwie modyfikacje powyższego algorytmu, polegające na zmianie współczynnika stojącego przed liczbą n , gdy jest ona nieparzysta. W pierwszej modyfikacji przyjmijmy, że zamiast przez 3 mnożymy liczbę n przez 2, a w drugim — że mnożymy ją przez 1. Oznaczmy te zmodyfikowane algorytmy odpowiednio: Ciąg-2 i Ciąg-1.

Wygeneruj algorytmem Ciąg-2 po dziesięć kolejnych elementów ciągu, gdy początkowe wartości liczby n są równe 6, 7 i 8. Nie powinieneś mieć trudności z wykonaniem następnego ćwiczenia.

Ćwiczenie 12.2. Wykaż, że algorytm Ciąg-2 kończy swoje działanie wtedy i tylko wtedy, gdy dana liczba n jest potęgą liczby 2. ■

Rozważmy teraz algorytm Ciąg-1. Jeśli liczba n jest parzysta, to maleje o połowę, a jeśli jest nieparzysta — to najpierw wzrasta o 1, a w następnym kroku, już jako liczba parzysta, maleje o połowę. Zatem w dwóch kolejnych krokach liczba n zmniejsza się o blisko połowę. O tym algorytmie można wykazać już coś więcej.

Zadanie 12.1. Uzasadnij szczegółowo, że algorytm Ciąg-1 kończy działanie dla każdej naturalnej liczby n . ■

Z badaniem skończoności obliczeń jest związany bardzo poważny problem informatyczny, tzw. **problem stopu**, który polega na stwierdzeniu, czy istnieje jeden algorytm, który mógłby być użyty do sprawdzania skończoności działania każdego algorytmu, również siebie. Dla konkretnego algorytmu i jego poprawnych danych jeśli algorytm kończy działanie, to dobrze, ale jeśli się nie zatrzymuje, to na ogół nie wiemy, kiedy możemy przerwać oczekiwanie na jego zakończenie. Niestety, problem stopu nie ma rozwiązania, a dokładniej — jest nierozstrzygalny.



Bardzo przystępnie jest to uzasadnione w artykule J. Nievergelta w kwartalniku „Komputer w Edukacji” nr 1/1994 (zobacz również w książce [Harel, str. 211]). ■

12.4. Złożoność i efektywność algorytmów

Wiele z naszych rozważań w tej książce uwzględnia sugestie Ralpha Gomory'ego (punkt 1.2), że najbardziej właściwym sposobem przyspieszania obliczeń komputerowych jest obarczanie ich mniejszą liczbą działań do wykonania. Naszym celem było więc opracowanie najlepszych algorytmów, czyli wykonujących jak najmniej działań. Posługujemy się przy tym pojęciem **złożoności obliczeniowej algorytmu**, którą określamy jako liczbę operacji wykonywanych przez algorytm. Ponieważ wszystkie prezentowane przez nas algorytmy są uniwersalne, złożoność jest wyznaczana w zależności od liczby danych, a czasem również od ich wielkości.

Staraliśmy się w tej książce podać dla każdego problemu możliwie najlepszy pod względem złożoności algorytm. Dla wielu problemów przedstawiamy bezwzględnie najlepsze algorytmy ich rozwiązywania — **algorytmy optymalne**. Są to tak popularne i często występujące problemy, jak: znajdowanie w różnych zbiorach elementów najlepszych lub najgorszych, jednoczesne znajdowanie jednych i drugich, porządkowanie różnych zbiorów, umieszczanie nowych elementów w już uporządkowanych zbiorach tak, aby nie psuć zastanego porządku.

Obok pojęcia złożoności obliczeniowej, używamy w tej książce również pojęcia pokrewnego — **efektywność algorytmu**, które odnosimy do złożoności w praktycznym sensie. Najczęściej pojawia się ono w porównaniach różnych metod rozwiązywania tego samego problemu lub w ocenie praktycznej przydatności algorytmów. Na przykład, możemy powiedzieć, że algorytm porządkowania przez wybór jest efektywniejszy od algorytmu bąbelkowego (choć oba algorytmy mają złożoność obliczeniową proporcjonalną do n^2) lub że nie ma efektywnego algorytmu rozwiązania problemu komiwojażera dla 100 miast.

Chcielibyśmy jeszcze raz podkreślić użyteczność naszych rozważań. Coraz większe wymagania stawiane człowiekowi i jego maszynom powodują, że stale poszukujemy coraz lepszych metod zmagania się z problemami. Nie mówiliśmy zbyt wiele o problemach, które są najtrudniejsze, zarówno dla człowieka, jak i nawet dla najszybszych komputerów. Przykładem tutaj może być problem komiwojażera (punkt 1.2 oraz problem 13.35). Może się to wydawać paradoksalne, ale dopiero rozwój algorytmiki na potrzeby coraz szybszych obliczeń uświadomił nam, iż dysponując obecną wiedzą, mamy małe szanse znalezienia optymalnego rozwiązania problemu komiwojażera dla praktycznych sytuacji. Cała nadzieja w tym, że algorytmika nie zapisała jeszcze do końca swoich najlepszych stron.

12.5. Zadania i problemy

Zadanie 12.2. Zmodyfikuj dowód poprawności algorytmu binarnego

przeszukiwania podany w tekście, aby otrzymać dowód poprawności algorytmu binarnego umieszczania (zobacz punkt 9.2).

Problem 12.1. Omawiając algorytm Euklidesa w punkcie 7.4, podaliśmy wszystkie informacje niezbędne do wykazania jego całkowitej poprawności, czyli zgodności działania ze specyfikacją oraz skończoności obliczeń. Sformułuj te fakty i użyj ich w dowodzie.

Problem 12.2. Wykaż całkowitą poprawność procedur EuklidDziel i EuklidOdej, które są realizacjami w języku Pascal obu wersji algorytmu Euklidesa, o których jest mowa w problemie 12.2. Teksty tych procedur znajdziesz wśród materiałów dostępnych na stronie książki: <http://edukacja.helion.pl/algorytmika>. Możesz wybrać odpowiednie realizacje algorytmu Euklidesa w języku Python.

Miałeś okazję jeszcze raz spojrzeć na poznane algorytmy i:

- ▶ sprecyzować własności, które musi mieć każdy algorytm — są wśród nich: **poprawność** (czyli zgodność ze specyfikacją), **dobra określoność** wszystkich działań, **uniwersalność**, **skończoność działania** i **efektywność**;
- ▶ sprawdzić, czy mają one te własności — gdyż to gwarantuje, że algorytmy **poprawnie rozwiązują** problemy, dla których zostały skonstruowane i mogą być **efektywnie stosowane**.

Rozdział 13. Problemy

W tym rozdziale zamieszczamy problemy do samodzielnego rozwiązania. Są one podzielone na dwie grupy. W pierwszej (punkt 13.1) znajdują się te, które można rozwiązać, korzystając z wiadomości i umiejętności nabytych podczas rozwiązywania ćwiczeń, zadań i problemów zamieszczonych w poprzednich rozdziałach. W drugiej grupie (punkt 13.2) zebraliśmy problemy trudniejsze, których rozwiązanie często wymaga wykonania pewnych czynności badawczych.

Większość problemów przedstawionych w tym rozdziale, zwłaszcza w drugiej grupie, zawiera materiał, który jest istotnym rozszerzeniem rozważań z poprzednich rozdziałów. Dlatego ich sformułowania są bardziej szczegółowe, a przez to obszerniejsze. Treść niektórych problemów jest uzupełniona dodatkowymi informacjami, wskazówkami oraz odnośnikami do innych fragmentów książki lub do innych opracowań — informacje te zamieszczamy w punkcie Uwagi, następującym bezpośrednio po treści problemu.

Większość problemów jest związanych z napisaniem odpowiedniego programu w języku Pascal lub Python.

13.1. Problemy łatwiejsze

Masz sposobność, na specjalnie dobranych przykładach, wykazać się umiejętnościami wykorzystania sposobów rozwiązywania problemów, nabytymi w trakcie lektury poprzednich rozdziałów. Algorytmy, jakie musisz tutaj zastosować, zostały wcześniej podane lub są ich niewielkimi modyfikacjami.

Znajdują się tutaj również problemy, których rozwiązanie polega na wyprowadzeniu dodatkowych własności omówionych wcześniej algorytmów.

Problem 13.1. W dobie komputerów wiele pań zaczęło nadawać swoim zeszytom z przepisami kulinarnymi tytuł *ALGORYTMY*. Zapewne ma na to wpływ pojawianie się takich przepisów, jako przykładów, w książkach poświęconych algorytmom — zobacz [Harel]. I chociaż najczęściej dyskutuje się w nich o niedostatkach przepisów kulinarnych jako algorytmów, Czytelnik może odnieść wrażenie, że ma do czynienia z pewną klasą algorytmów.

Postaraj się zebrać argumenty, by uzasadnić, dlaczego przepisy kulinarne na ogół nie są dobrymi przykładami algorytmów w rozumieniu tej książki.

Wybierz z dostępnej Ci książki kucharskiej przykłady dwóch przepisów, jeden — który potrafisz przeformułować jako algorytm w postaci listy kroków, drugi — który się temu nie poddaje. W obu przypadkach podaj uzasadnienie.

Rozważ również etap „implementacji” (wdrożenia), czyli wykonania algorytmów kulinarnych, i zastanów się nad jednoznacznością określenia poszczególnych kroków (takich np., jak „do smaku dodaj szczyptę soli”) i jednoznacznością „wyniku końcowego”. W dalszej kolejności wykonaj „eksperyment” z wybranymi algorytmami w celu potwierdzenia swoich obserwacji i wniosków.

Uwaga. Szczegółowe rozwiązanie tego problemu przedstawiamy w książce [Piramidy], w rozdziale 1., na przykładzie chłodnika litewskiego.

Problem 13.2. W punkcie 1.2 wspominamy o problemie, który polega na znalezieniu najkrótszej trasy zamkniętej, przechodzącej przez każdy punkt ustalonego zbioru n elementów dokładnie jeden raz. Wszystkich takich tras jest $(n - 1)!$ W rozdziale 4. wyprowadzamy, ile jest różnych uporządkowań 3, 4, 5 i 6 liczb.

Uogólnij tamto rozumowanie i wykaż, że liczba wszystkich możliwych uporządkowań n liczb jest równa $n! = 1 \cdot 2 \cdot \dots \cdot n$ (zobacz zadanie 8.3).

Problem 13.3. Drzewo rozgrywek w turnieju tenisowym przypomina drzewo obliczania wartości wyrażenia, zdefiniowane w punkcie 1.3.6: w wierzchołkach końcowych znajdują się dane (w turnieju — zawodnicy do uszeregowania), a w wierzchołkach pośrednich — działania (porównania zawodników, czyli mecze między nimi, wraz ze zwycięzcą meczu). Zatem znajdowanie największego elementu w zbiorze można traktować jak obliczanie wartości funkcji, która ma być równa wartości największej spośród danych liczb. Narysuj drzewo obliczania wartości tej funkcji, realizujące algorytm Max. Podaj również sposób znajdowania drugiej największej liczby, w którym korzysta się z wyników pierwszego etapu znajdowania największej liczby. Ile porównań należy wykonać w tym przypadku, by znaleźć drugi największy element wśród danych?

Problem 13.4. W punkcie 7.5.2 są podane wzory ułatwiające wykonywanie czterech podstawowych działań na ułamkach zwykłych. Umożliwiają one redukowanie dużych wartości liczb występujących w obliczeniach dzięki wczesnemu skracaniu ich przez odpowiednie dzielniki wspólne.

(A) Napisz zestaw funkcji w języku Pascal lub w języku Python, realizujących podstawowe operacje na ułamkach zwykłych według algorytmów opisanych w punkcie 7.5.2.

(B) W niektórych komputerowych realizacjach języka Pascal i Python nie zawsze jest sygnalizowane wystąpienie nadmiaru stałopozycyjnego, tzn. obliczenia nie są przerywane, gdy ich wyniki wykraczają poza zakres realizacji danego typu liczb całkowitych. W związku z tym, dla używanej przez Ciebie realizacji języka Pascal lub Python i dla każdego z zakresów liczb typu integer, określ zakres ułamków, w jakim Twoje funkcje wykonują wszystkie działania poprawnie — tzn. ani wyniki pośrednie, ani końcowe nie wykraczają poza zakres reprezentacji.

Problem 13.5. Czy algorytm porządkowania przez wybór w podanej przez nas

realizacji jest algorytmem stabilnym? A czy można tę własność zmienić na przeciwną — tzn., czy można ten algorytm zrealizować zarówno jako stabilną, jak i niestabilną metodę porządkowania?

Problem 13.6. Co jest wartością następującej funkcji CoToJest dla dowolnej liczby naturalnej n ?



```
function CoToJest(n:integer):integer;
  var i,k,l:integer;
begin
  if n <= 2 then CoToJest := 1
  else begin
    k := 1; l := 1;
    for i := 1 to (n div 2)-1 do begin
      k := k + l; l := l + k
    end; {for}
    if odd(n) then l := l + k;
    CoToJest := l
  end {if n > 2}
end; {CoToJest}
```



```
def CoToJest(n):
  if n <= 2:
    return 1
  else:
    k = 1
    l = 1
    for i in range(1,(n // 2)):
      k = k + l
      l = l + k
    if n % 2 == 1:
      l = l + k
    return l
```

Uwagi. Zapewne, udzielenie odpowiedzi na powyższe pytanie nie sprawi Ci większego kłopotu. Jeśli tak, to porównaj tę funkcję z innym, iteracyjnym

sposobem obliczania tych samych wielkości, podanym w tekście. Jaka jest różnica między tymi dwoma sposobami pod względem liczby iteracji i liczby wykonywanych elementarnych operacji arytmetycznych? Aby potwierdzić swoje obserwacje, wykonaj eksperyment obliczeniowy za pomocą komputera.

Problem 13.7. Jaka jest następna liczba w ciągu: 1, 1, 2, 4, 7, 13, 24, 44, 81? Podaj zależność, według której można wyznaczać te liczby. Opisz algorytm generowania tych liczb, iteracyjny i rekurencyjny.

Uwaga. Jako wskazówkę przyjmij, że elementy tego ciągu można by nazwać liczbami Tribonacciego.

Problem 13.8. Opracuj algorytm, który dla dziesiętnej liczby naturalnej n znajduje kolejne jej cyfry w systemie o podstawie b , od najbardziej znaczącej począwszy. Następnie opisz go w języku Pascal lub Python.

Uwagi. Dla podstawy $b = 2$ podaliśmy w punkcie 7.1 iteracyjny algorytm znajdowania binarnego rozwinięcia liczby dziesiętnej, który generuje jednak kolejne bity od najmniej znaczącego począwszy. Aby otrzymać rozwinięcie rozpoczynające się od najbardziej znaczącego bitu, zaproponowaliśmy tam odwrócenie reprezentacji otrzymanej w wyniku działania tego algorytmu (zobacz problem 7.2).

Do rozwiązania tego problemu możesz skorzystać z rekurencji. Zauważ przy tym oczywistą własność, że cyfra jedności w reprezentacji liczby n jest poprzedzona przez cyfry bardziej znaczące. Sformułuj tę własność jako zależność rekurencyjną i nie zapomnij o warunku zakończenia rekurencji. Użyj w tej zależności funkcji `div` i `mod`.

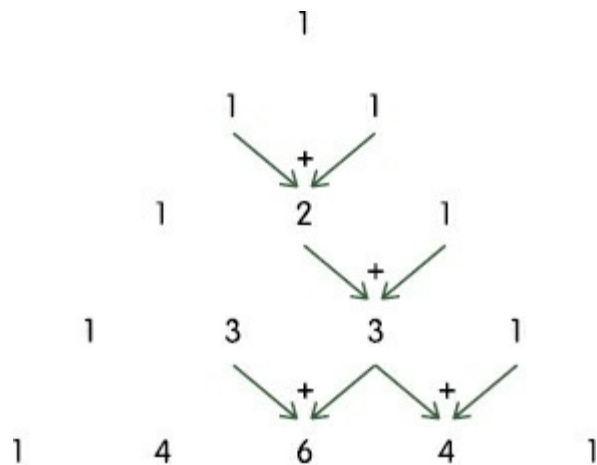
Rozwiązanie tego problemu można podpatrzeć w książce [Piramidy], w rozdziale 5.

Problem 13.9. Na rysunku 13.1 jest przedstawiona tablica liczb, będąca fragmentem **trójkąta Pascala**. Wiersze w tej tablicy są ponumerowane od 0. Wiersz o numerze n , dla $n \geq 1$, jest ciągiem współczynników przy kolejnych składnikach w rozwinięciu potęgi $(a + b)^n$, zwanej **dwumianem Newtona** — składniki te numerujemy od 0, co 1, aż do n . Sprawdź, że jest to prawdą dla $n = 1, 2, 3$. Oznaczmy przez $C_{n, k}$ liczbę o numerze k w wierszu n . Mamy zatem dla sześciangu sumy: $C_{3, 0} = 1, C_{3, 1} = 3, C_{3, 2} = 3, C_{3, 3} = 1$.

Zauważ następującą zależność między liczbami $C_{n, k}$: najbardziej krańcowe liczby w każdym wierszu są równe 1, a każda inna liczba jest równa sumie liczb stojących nad nią w poprzednim wierszu. Tę obserwację dla każdego wiersza $n = 1, 2, \dots$ można zapisać w następujący sposób:

$$C_{n, 0} = C_{n, n} = 1$$

$$C_{n, k} = C_{n-1, k-1} + C_{n-1, k}, \text{ dla } 0 < k < n \quad (13.1)$$



Rysunek 13.1. Trójkąt Pascala

Zależności (13.1) mają postać rekurencyjną z dwoma indeksami, n i k .

(A) Opisz, w postaci listy kroków, rekurencyjny algorytm tworzenia trójkąta Pascala według wzorów (13.1) i napisz funkcję w języku Pascal lub Python będącą realizacją tego algorytmu.

(B) Opisz, w postaci listy kroków, iteracyjny algorytm tworzenia trójkąta Pascala i napisz funkcję w języku Pascal lub Python będącą realizacją tego algorytmu.

Problem 13.10. Problem geometryczny. Danych jest n prostych: $y_i(x) = a_i x + b_i$ dla $i = 1, 2, \dots, n$, z których żadne dwie nie przecinają się w przedziale domkniętym $[0, 1]$. Podaj algorytm, który dla punktu (x, y) , gdzie $0 \leq x \leq 1$, znajduje dwie sąsiednie proste, między którymi leży ten punkt — lub sygnalizuje, że ten punkt leży poniżej lub powyżej prostych.

Uwaga. Ten problem można nazwać **przeszukiwaniem dwuwymiarowym**.

Problem 13.11. Zapisz algorytm przeszukiwania binarnego (podany w punkcie 9.2) w postaci algorytmu rekurencyjnego i utwórz jego realizację w języku Pascal lub Python.

Problem 13.12. Podaj opis algorytmu porządkowania przez scalanie w postaci iteracyjnej.

Uwaga. Najpierw możesz rozważyć łatwiejszą wersję tego problemu (zobacz zadanie 10.4), w której długość porządkowanego ciągu jest potęgą liczby 2.

Problem 13.13. Napisz program, w którym umieścisz obie realizacje porządkowania przez scalanie: rekurencyjną (punkt 10.2) i iteracyjną, będącą przedmiotem poprzedniego problemu. Twój program ma porównywać efektywność obu realizacji na dość długich ciągach, których elementy generuj losowo. Użyj również ciągów malejących oraz rosnących.

Problem 13.14. W rozdziale 4. przedstawiliśmy algorytm porządkowania 5 liczb, w którym jest wykonywanych co najwyżej 7 porównań. Sprawdź, ile

porównań w przypadku 5 liczb wykonują takie algorytmy porządkowania, jak: bąbelkowy, przez wybór, przez umieszczanie, przez scalanie oraz szybki.

Problem 13.15. Wypełnij tabelę 13.1, wstawiając w wolne pola liczby porównań, wykonywanych w wymienionych algorytmach na ciągach o długości n , złożonych z liczb spełniających dodatkowe warunki.

Tabela 13.1. Złożoności wybranych algorytmów, zastosowanych do szczególnych ciągów liczb o długości n

Algorytmy porządkowania	Rodzaje ciągów		
	Równe elementy	Uporządkowany rosnąco	Uporządkowany malejąco
bąbelkowy			
przez wybór			
przez umieszczanie			
przez scalanie			
szybki			

Problem 13.16. Zapisz rekurencyjny algorytm znajdowania wyjścia z labiryntu, podany w punkcie 11.1.2, w postaci algorytmu iteracyjnego.

Uwaga. Jak w większości iteracyjnych realizacji algorytmów rekurencyjnych, tak i w tym przypadku musisz się posłużyć listą kolejno przechodzonych pól — tworzy ona **stos**, tj. pola są dokładane do tej listy jedno na drugie, a pobierane od góry, czyli najpierw jest pobierane to pole, które było dołożone na końcu.

Problem 13.17. W zadaniu 11.4 opisaliśmy problem wydawania reszty możliwie najmniejszą liczbą banknotów i monet. Uzasadnij, że jeśli nominały mają wartości $d_{k-1}, d_{k-2}, \dots, d^1, d^0$, dla pewnych liczb naturalnych $d > 1$ oraz $k \geq 0$, to zachłanny algorytm podany w tamtym zadaniu zawsze tworzy reszty złożone z najmniejszej liczby banknotów i monet.

Uwaga. Zapisz resztę M , którą masz utworzyć, jako wyrażenie zależne od nominałów $d_{k-1}, d_{k-2}, \dots, d^1, d^0$ i skorzystaj z tej postaci liczby M — co Ci ta postać przypomina?

Problem 13.18. Dana jest prostokątna tablica o wymiarach m wierszy na n kolumn, złożona z liczb naturalnych (tabela 13.2.) Załóżmy, że jej wiersze są ponumerowane od góry do dołu, a kolumny — od lewej do prawej. W tej tablicy można poruszać się jedynie w prawo lub do dołu. Drogą w takiej tablicy

nazywamy ciąg pól, w którym każde dwa sąsiednie pola mają wspólny bok i spełnione jest założenie o kierunku poruszania się, a więc drogą w przykładowej tablicy jest np. ciąg pól: $(1,1), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5)$. Długością takiej drogi jest suma liczb znajdujących się w polach tworzących tę drogę. Twoim zadaniem jest znaleźć najdłuższą drogę z pola w lewym górnym rogu do pola w prawym dolnym rogu, czyli z pola $(1, 1)$ do pola (m, n) . Przykładowa droga jest właśnie drogą z $(1, 1)$ do $(4, 5)$, a jej długość wynosi $1 + 8 + 8 + 5 + 2 + 1 + 3 + 4 = 32$ — nie jest to jednak najdłuższa droga z $(1, 1)$ do $(4, 5)$.

Tabela 13.2. Prostokątna tablica z liczbami naturalnymi

1	2	3	4	5
1	1	3	7	2
2	8	2	4	8
3	8	4	9	7
4	5	2	1	3

Podaj algorytm programowania dynamicznego, który rozwiązuje ten problem dla dowolnej tablicy o wymiarach m na n .

Uwaga. Wykorzystaj spostrzeżenie, że do danego pola w tablicy, w szczególności do końcowego pola drogi, można dojść jedynie z lewej strony lub od góry. Podpowiedzi do rozwiązania tego problemu można znaleźć w rozdziale 15. w książce [Piramidy].

13.2. Problemy trudniejsze

Masz teraz okazję wykazać się umiejętnościami rozwiązywania trudniejszych problemów. Algorytmy, jakie masz tutaj opracować i stosować, nie były wcześniej przez nas wyprowadzone, chociaż do ich utworzenia możesz się posłużyć metodami, z których korzystaliśmy w innych algorytmach.

Powodzenia w łamaniu głowy!!!

Problem 13.19. Często w obliczeniach występuje suma elementów, w której co drugi element jest dodawany, a co drugi — odejmowany, na przykład $1/2 - 1/3 + 1/4 - 1/5 + \dots + (-1)^n/n$. Jeśli miałbyś zaproponować metodę obliczania wartości tego wyrażenia, to zapewne pierwszym pomysłem, jaki przyszedłby Ci do głowy, byłoby obliczenie sumy elementów dodatnich, potem sumy elementów

ujemnych, a następnie odjęcie tej drugiej liczby od tej pierwszej. Zastanów się jednak, jak można obliczyć wartość takiej sumy, wykonując tylko jeden przebieg sumowania, za pomocą jedynie dodawania lub odejmowania, i bez sprawdzania żadnego warunku.

Uwaga. Rozważ obliczanie sumy elementów o naprzemiennych znakach od końca.

Problem 13.20. Masz obliczyć sumę dużej ilości liczb. Liczby te mogą przyjmować wartości o dużej rozpiętości. Interesuje nas wynik podany z dokładnością do czterech najbardziej znaczących cyfr.

Przypomnijmy najpierw, na czym polega wykonanie działania (w tym problemie — dodawania) z ustaloną dokładnością do czterech cyfr:

► wszystkie liczby występujące w obliczeniach mają co najwyżej cztery cyfry znaczące (np. 10 000 powinniśmy zapisać w postaci $1E4$, tj. $1 \cdot 10^4$, dalej jednak będziemy pisać 10 000, pamiętając że ostatnia cyfra w tej liczbie, czyli zero, nie ma znaczenia);

► dodanie dwóch liczb polega na obliczeniu ich sumy i zaokrągleniu wyniku do czterech cyfr, czyli wynik dodawania $10\,000 + 1$ wynosi 10 000, gdyż we właściwym wyniku 10 001 zaokrąglonym do czterech cyfr nie jest uwzględniana cyfra 1 na końcu; ponadto, wykonywanie ciągu działań polega na wykonaniu poszczególnych działań zgodnie z tą zasadą, czyli zaokrągleniu każdego wyniku pośredniego.

Przykład 13.1. Mamy obliczyć sumę jedenastu liczb: $10\,000 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$. Wynik niestety będzie równy 10 000, gdyż po każdym dodawaniu wynik jest zaokrąglany do 10 000, chociaż ta suma zaokrąglona do czterech cyfr wynosi 10 010. Nawet gdybyśmy w ten sposób dodawali do liczby 10 000 aż dziesięć tysięcy jedynek, to wynik pozostanie równy 10 000. ■

Twoim zadaniem jest teraz zaproponować algorytm obliczania, z dokładnością do czterech cyfr, sumy ciągu liczb o różnych wartościach, w którym według Ciebie nie dochodzi do utraty dokładności (spowodowanej dużą różnicą tych wartości) lub aby ta utrata była możliwie najmniejsza.

W algorytmie, na który Ciebie naprowadzamy w rozwiązaniu problemu 13.20, oszacowanie błędu zaokrągleń wyniku jest najmniejsze w porównaniu z oszacowaniem uzyskiwanym w innych algorytmach obliczania sumy liczb. Można więc uznać, że daje on najdokładniejszy wynik dodawania.

Autor przewiduje Twoje pierwsze rozwiązanie, które zapewne łatwo wyprowadzisz z podanego wyżej przykładu. Jaki otrzymasz wynik, obliczając sumę liczby 10 000 i dwudziestu tysięcy jedynek? Jeśli wynosi ona 20 000, to namyśl się głębiej, czy nie możesz jeszcze bardziej usprawnić swojego algorytmu tak, aby wynik był bliski rzeczywistej sumy, czyli bliski 30 000.

Przedstaw, w postaci listy kroków, swój ostateczny algorytm dla tego problemu. Uwzględnij w opisie sposób przechowywania danych i wyników pośrednich, czyli odpowiednie struktury danych. Zapisz ten algorytm w języku Pascal lub Python. Polecamy tutaj rozdział 13 w książce [Piramidy].

Problem 13.21. Drzewa wyrażeń (zobacz punkt 1.3.6) mogą być wykorzystane do interpretacji działania algorytmów znajdowania największych i najmniejszych elementów w zbiorze. Takie drzewo przyporządkowane jakimukolwiek algorytmowi, w którym są wykonywane tylko porównania między elementami zbioru, ma n wierzchołków końcowych (liści), gdzie n jest liczbą elementów w zbiorze. A ile ma wierzchołków pośrednich, czyli ile jest wykonywanych porównań? Wykaż indukcyjnie, że każde takie drzewo o n wierzchołkach końcowych ma $n - 1$ wierzchołków pośrednich.

Problem 13.22. W większym towarzystwie, zwłaszcza na oficjalnym przyjęciu, na ogół nie wszyscy się znają. Co więcej, mogą się pojawić osoby powszechnie znane, ale niekoniecznie znające tych, którzy ich znają. **Idolem** możemy nazwać osobę, która jest znana przez wszystkich z towarzystwa, ale nie zna nikogo. Twoim zadaniem jest wykryć, czy w towarzystwie złożonym z n osób znajduje się idol. Jediną „operacją”, jaką możesz wykonywać, jest: wybrać dwie osoby i jednej z nich zadać pytanie, czy zna tę drugą osobę.

Problem odszukania idola wśród n osób odegrał znaczącą rolę w algorytmice. Jest bowiem przykładem problemu, dla którego zbiór danych składa się z $n(n - 1)$ bitów informacji (tymi informacjami są „czy jedna osoba zna drugą osobę”), a może być rozwiązany po sprawdzeniu jedynie $3(n - 1)$ z nich. Zauważmy, że pierwsza liczba jest funkcją kwadratową względem n (ma postać $n^2 - n$), a druga — jest funkcją liniową. Zatem o tym problemie można powiedzieć, że nie wszystkie dane są istotne do jego rozwiązania, gdyż możemy zastosować algorytm, który korzysta tylko z niewielkiej ich części.

Potencjalnie spośród n osób możesz wybrać $n(n - 1)/2$ par i zadać $n(n - 1)$ pytań (zauważ, że każdej parze możesz zadać dwa pytania, po jednym każdej z osób). Nie chodzi nam jednak o taki sposób, w którym pytane są niemal wszystkie pary. Istnieje algorytm wykrywania idola wśród n osób, który działa w podobny sposób, jak algorytm znajdowania lidera opisany w punkcie 5.6.1: w pierwszym etapie eliminuje się osoby, które nie mogą być idolem, a w drugim — sprawdza się, czy jedyny kandydat na idola jest nim rzeczywiście. Podaj szczegółowy opis takiego algorytmu i określ, ile wykonywanych jest w nim podstawowych „operacji”, czyli ile zadajesz pytań. Zrealizuj ten algorytm w wybranej przez siebie reprezentacji.

Uwagi. Do znalezienia algorytmu, o który pytamy, wystarczy zauważyć, że w wyniku wykonania jednej „operacji”, z dwóch osób zawsze pozostaje jedna jako kandydat na idola. W algorytmie, o który nam chodzi, nie powinniśmy wykonywać więcej niż $3(n - 1)$ „operacji”.

Problem 13.23. Dany jest ciąg n liczb rzeczywistych x_1, x_2, \dots, x_n , mogą wśród nich być również liczby ujemne. Podaj algorytm znajdowania w tym ciągu spójnego fragmentu, czyli podciągu złożonego z pewnej ilości kolejnych liczb x_k, x_{k+1}, \dots, x_l (gdzie $1 \leq k \leq l \leq n$) o największej możliwej sumie elementów.

Zapewne pierwszym pomysłem rozwiązania tego problemu będzie przeglądanie wszystkich możliwych spójnych podciągów ciągu danych — takich podciągów jest tyle, ile uporządkowanych par indeksów k i l , spełniających nierówności $1 \leq k \leq l \leq n$. W tym wymyślonym *ad hoc* algorytmie liczba iteracji będzie proporcjonalna do n^2 , w każdej iteracji są sumowane $l - k + 1$ liczby. Istnieje jednak algorytm, który znajduje rozwiązanie tego problemu w czasie proporcjonalnym do liczby n , jednokrotnie tylko przeglądając ciąg danych. Podamy to rozwiązanie, gdyż nie jest łatwo wpaść na jego trop. Przedstawimy je w postaci funkcji MaxSuma, zapisanej w języku Pascal i Python.



```
function MaxSuma(n:integer; x:Tablica1nr):real;
  var i :integer;
      Max,MaxI:real;
begin
  Max := 0; MaxI := 0;
  for i := 1 to n do
    if MaxI+x[i] > 0 then begin
      MaxI := MaxI+x[i];
      if MaxI > Max then Max := MaxI
    end
    else MaxI := 0;
  MaxSuma := Max
end; {MaxSuma}
```



```
def MaxSuma(n):
  Max = 0
  MaxI = 0
  for i in range(0,n):
    if MaxI+x[i] > 0:
      MaxI = MaxI + x[i]
    if MaxI > Max:
```

Max = MaxI

else:

MaxI = 0

return Max

Uzasadnieniem poprawności działania algorytmu zrealizowanego w tej funkcji jest następująca obserwacja: największa suma elementów spośród i początkowych elementów ciągu jest albo największą sumą wśród pierwszych $i - 1$ elementów (w funkcji oznaczona przez Max), albo jest sumą elementów podciągu kończącego się na pozycji i (w funkcji oznaczona przez MaxI).

Sprawdź najpierw, wykonując odpowiednie przykłady, a później uzasadnij dokładnie, że algorytm zrealizowany w funkcji MaxSuma jest poprawny. Sprawdzianem, czy dobrze rozumiesz przedstawione rozwiązanie, może być następujące zadanie: zmodyfikuj funkcję MaxSuma tak, aby dodatkowo były w niej określane indeksy początku i końca spójnego podciągu o największej sumie.

Uwagi. Algorytm dla tego problemu nie jest trywialny, podaliśmy więc go, a Twoim zadaniem jest jedynie zrozumienie, na czym polega to rozwiązanie.

Szczegółowy opis wyprowadzenia tego algorytmu, od metody wspomnianej na początku do algorytmu zrealizowanego w funkcji MaxSuma, jest zamieszczony w książce [Bentley, rozdział 7.]. Problem ten ma wiele zastosowań w zagadnieniach rozpoznawania wzorca i w problemach optymalizacyjnych.

Problem 13.24. W pliku lub w liście jest dany co najmniej trzelementowy zbiór A liczb naturalnych, będących długościami odcinków (takie dane możesz sam przygotować). Opisz algorytm, który sprawdza, czy z każdych trzech odcinków z tego zbioru można zbudować trójkąt.

Przypomnij sobie najpierw **warunek trójkąta**, czyli warunek, jaki muszą spełniać trzy liczby, aby były długościami boków trójkąta. Oczywiście, najprościej jest go sprawdzić dla każdych trzech liczb spośród danych. Ale czy rzeczywiście jest to najprostszy algorytm? Określ, ile wykonasz w nim działań (porównań)? Zauważ, że w takim algorytmie plik z danymi będzie musiał być wielokrotnie przeglądany w poszukiwaniu kolejnej trójki liczb do sprawdzenia warunku trójkąta. Stawiamy Ci wyzwanie: poszukaj algorytmu, który tylko raz przegląda plik z danymi i po wybraniu odpowiedniej trójki odcinków tylko... jeden raz sprawdza warunek trójkąta.

Uwaga. Pełna dyskusja rozwiązania tego problemu, w trochę bardziej złożonej postaci, jest zamieszczona w materiałach z I Olimpiady informatycznej [OI] oraz w poradniku [EI-III, rozdział 7.].

Problem 13.25. Zbiór do uporządkowania składa się z n elementów, które należy ustawić względem wartości ich cech, będących liczbami naturalnymi k_i ($i = 1, 2, \dots, n$) z ustalonego przedziału $[p, q]$. W porównaniu z liczbą elementów w

zbiornie długość tego przedziału jest niewielka. Oczywiście w takim przypadku różne elementy mogą mieć takie same cechy. Podaj algorytm porządkowania elementów takiego zbioru względem ich cech, w którym wykorzystuje się wartości tych cech, ale ich nie porównuje. Powinien on natomiast działać szybciej niż jakikolwiek algorytm wykonujący porównania elementów.

Problem 13.26. Przedstawmy schemat Hornera (7.6) w postaci, w której kolejne wartości zmiennej y są oznaczone zmienną z indeksem:

$$b_0 := a_0$$

$$b_i := b_{i-1}z + a_i, i = 1, 2, \dots, n.$$

Wtedy wartość ostatniej zmiennej jest równa wartości wielomianu, czyli $b_n = w_n(z)$. Wielkości b_i mają ciekawą interpretację.

(A) Wykaż, że prawdziwa jest następująca równość, w której występują współczynniki b_i (pomnóż obie jej strony przez $x - z$, pogrupuj wyrazy i porównaj obie strony):

$$\frac{w_n(x)}{x - z} = b_0x^{n-1} + b_1x^{n-2} + \dots + b_{n-1} + \frac{b_n}{x - z} \quad (13.2)$$

Oznaczmy

$$w_{n-1}(x) = b_0x^{n-1} + b_1x^{n-2} + \dots + b_{n-1}.$$

Wzór (13.2) można odczytać następująco: gdy podzielimy wielomian $w_n(x)$ przez dwumian $x - z$, otrzymamy iloraz, który jest wielomianem stopnia $n - 1$ oraz resztę b_n . Zatem wyrazy b_i są współczynnikami ilorazu, a ostatni z tych wyrazów — jest resztą z tego dzielenia.

(B) Korzystając z równości (13.2), wykaż, że spełniona jest równość:

$$w_n'(z) = w_{n-1}(z),$$

czyli wartość pochodnej $w_n'(z)$ wielomianu $w_n(x)$ w punkcie z jest równa wartości wielomianu $w_{n-1}(x)$ w tym samym punkcie.

(C) Korzystając z wyników punktów (A) i (B), podaj algorytm jednoczesnego obliczania wartości wielomianu i jego pochodnej.

Problem 13.27. W **metodzie czynników** obliczania wartości potęgi x^m korzysta się z rozkładu wykładnika m na czynniki. Załóżmy, że wykładnik potęgi można przedstawić w postaci $m = pq$, gdzie p jest najmniejszą liczbą pierwszą dzielącą m i $q > 0$. Wtedy mamy $x^m = (x^p)^q$, czyli wartość x^m można obliczyć, podnosząc najpierw x do potęgi p , a następnie x^p do potęgi q . Jeśli m jest liczbą pierwszą, to przedstawiamy potęgę jako $x^m = x^{m-1}x$ (wtedy $m - 1$ jest liczbą parzystą, czyli podzielną przez przynajmniej 2) i obliczamy w ten sposób

xm^{-1} .

Ile mnożeń wykonasz tą metodą dla $m = 15$? Porównaj ją w tym przypadku z metodą binarną. Wykonaj obliczenia dla innych wartości wykładnika, np. dla $m = 13, 23, 25, 31, 33, 54$.

Przedstaw opis algorytmu realizującego metodę czynników. Skorzystaj w nim z algorytmu znajdowania najmniejszego czynnika pierwszego liczby naturalnej.

Uwaga. Szybki sposób potęgowania jest wszechstronnie omówiony w książce [Knuth-2, punkt 4.6.3].

Problem 13.28. Podaj algorytm obliczania wartości potęgi x^{23} , w którym wykonujemy 6 mnożeń.

Uwaga. Uprzedzamy (ale warto to również sprawdzić), że we wszystkich podanych przez nas sposobach obliczania wartości tej potęgi trzeba wykonać przynajmniej 7 mnożeń. Ponownie odsyłamy tutaj do książki [Knuth-2].

Problem 13.29. Rozważ następującą modyfikację zdolności rozmnażania się królików w pytaniu Fibonacciego (punkt 8.2.2): nowa para królików staje się płodna po miesiącu życia i rodzi jedną parę nowych królików, a potem staje się bezpłodna. Oznaczmy przez G_n liczbę par królików po n miesiącach. Napisz odpowiednią zależność rekurencyjną dla G_n . Wypisz wartości kolejnych liczb tego ciągu, przyjmując różne wartości dwóch pierwszych wyrazów, np. $G_0 = G_1 = 1$, $G_0 = 1$ i $G_1 = 2$, $G_0 = 2$ i $G_1 = 1$. Na podstawie tych przykładów sformułuj wniosek, jaką postać ma ciąg liczb G_n w zależności od warunków początkowych.

Problem 13.30. Liczby Fibonacciego, oprócz wzoru (8.8), spełniają jeszcze wiele innych zależności rekurencyjnych. Szczególnie pożyteczne są następujące z nich:

$$F_{2k} = F_k F_{k+1} + F_{k-1} F_k \quad (13.3)$$

$$F_{2k+1} = F_{k+1}^2 + F_k^2$$

w których liczba Fibonacciego z danym indeksem jest określona za pomocą liczb Fibonacciego z indeksami prawie o połowę mniejszymi. Opisz rekurencyjny i iteracyjny algorytm obliczania wartości F_n na podstawie wzorów (13.3). Porównaj ten algorytm z algorytmami wynikającymi z zależności (8.8) pod względem liczby wykonywanych działań.

Polecamy rozdział 6. w książce [Piramidy], w którym przedstawiono m.in. różne algorytmy obliczania wartości liczb Fibonacciego.

Problem 13.31. W arytmetyce często posługujemy się bardzo prostymi zależnościami, które prowadzą do bardzo efektywnych obliczeń komputerowych. Jedną z takich grup zależności stanowią tak zwane **zależności**

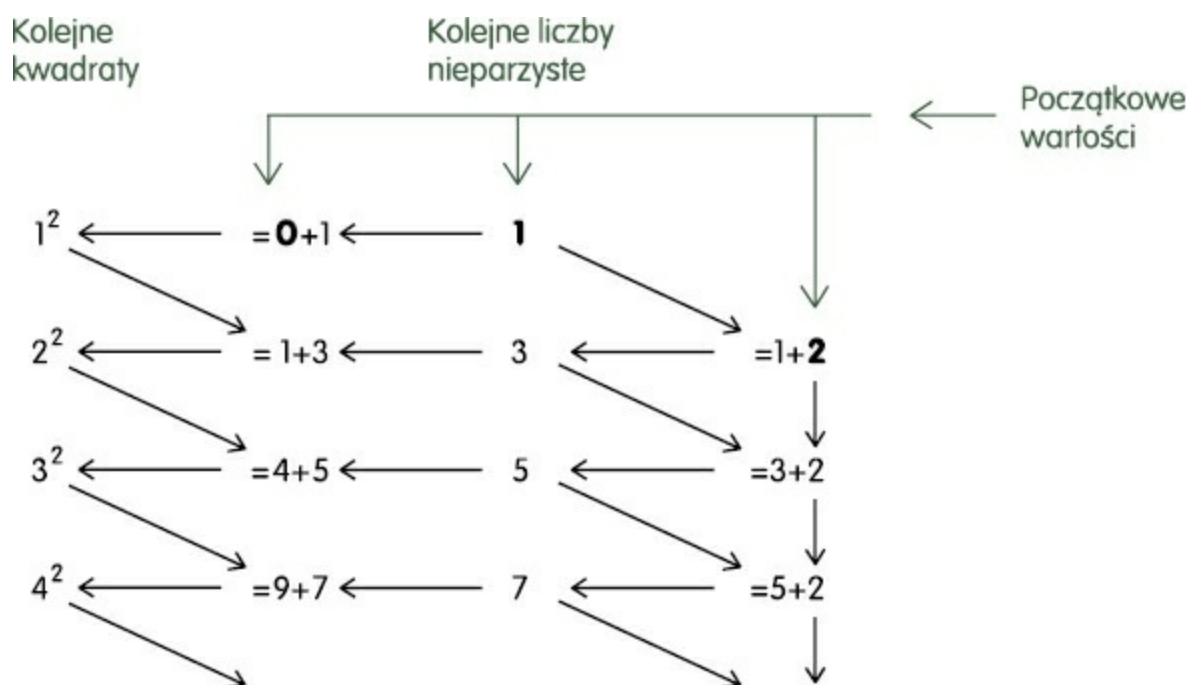
różnicowe, a przykładem jest następująca równość:

$$(n + 1)^2 - n^2 = 2n + 1.$$

Zależności różnicowe są bardzo przydatne przy tworzeniu tablic wartości funkcji obliczanych w równoodległych punktach — zajrzyj do swoich tablic matematycznych i przekonaj się, że właśnie takie są. Takimi obliczeniami interesował się Charles Babbage, który w pierwszej połowie XIX wieku zaprojektował i częściowo zbudował specjalną maszynę do wykonywania obliczeń według zależności różnicowych, zwaną **maszyną różnicową** (zobacz ilustrację tej maszyny w punkcie 1.2). W pełni funkcjonująca różnicowa maszyna Babbage’a, według jego projektu, została zbudowana dopiero w czasach maszyn elektronicznych na 200. rocznicę jego urodzin w 1991 roku, a samego Babbage’a uważa się dzisiaj za prekursora budowniczych komputerów.

Oznacza ona, że różnica między wartościami funkcji kwadratowej dla dwóch kolejnych liczb naturalnych jest równa kolejnej liczbie nieparzystej. Tę równość można również odczytać tak, że kwadrat kolejnej liczby naturalnej $n + 1$ jest równy sumie kwadratu poprzedniej liczby naturalnej n i kolejnej liczby nieparzystej $2n + 1$. Jest to więc również zależność rekurencyjna, na podstawie której kwadrat kolejnej liczby naturalnej może być wyznaczony za pomocą dwóch dodawań zamiast jednego mnożenia — w wielu komputerach trwa to krócej.

Na rysunku 13.2 jest przedstawiony schemat, według którego powinny przebiegać obliczenia generujące kwadraty kolejnych liczb naturalnych. Zapisz, w wybranej reprezentacji, algorytm wyznaczania kwadratów kolejnych liczb naturalnych, który jest realizacją tego schematu.



Rysunek 13.2. Schemat algorytmu obliczania kwadratów kolejnych liczb naturalnych. Danymi są liczby 0, 1 i 2, a każdy nowy kwadrat wyznaczamy,

wykonując dwa dodawania: by obliczyć kolejną liczbę nieparzystą i dodać ją później do poprzedniego kwadratu

Problem 13.32. W każdym z omówionych przez nas algorytmów porządkowania elementy są przestawiane, gdy zajmują niewłaściwe miejsca. W wielu praktycznych sytuacjach taka operacja może być bardzo trudna do przeprowadzenia lub wręcz niemożliwa — wyobraź sobie na przykład porządkowanie w komputerowej bazie danych kartotek pracowników dużego przedsiębiorstwa według ich stażu pracy. Wówczas algorytm porządkowania nie powinien przenosić rekordów (kartotek) z danymi.

Tę trudność można pokonać, wprowadzając tzw. wskaźniki do elementów, które mamy uporządkować. Oznaczmy przez p_i wskaźnik elementu i . Użycie wskaźników polega na tym, że jeśli mamy uporządkować ciąg elementów x_1, x_2, \dots, x_n , to w porównaniach występuje element x_{p_i} zamiast elementu x_i i jeśli

mamy przestawić element x_i , to przestawiamy jego wskaźnik p_i . Po zakończeniu algorytmu p_1 jest wskaźnikiem najmniejszego elementu, p_2 — wskaźnikiem drugiego najmniejszego elementu itd.

Wybierz jeden ze znanych Ci algorytmów porządkowania i podaj jego opis, wykorzystując opisane powyżej wskaźniki, dzięki czemu elementy porządkowanego ciągu nie muszą być przestawiane. Zrealizuj ten algorytm w języku Pascal lub Python.

Problem 13.33. Dany jest zbiór czynności $S = \{1, 2, \dots, n\}$ do wykonania. Czynność i rozpoczyna się w chwili s_i , a kończy przed chwilą f_i . Czynności te są wykonywane z użyciem tych samych narzędzi, a więc wykonywanie żadnych dwóch czynności nie może odbywać się równocześnie. Zaproponuj zachłanny algorytm znajdowania największego podzbioru czynności w zbiorze S , które będą mogły być wykonane. Sprawdź swój algorytm na dobranych przez siebie danych.

Uwaga. Rozważ, jaką wybrałbyś czynność, aby pozostało jak najwięcej czasu do wykonania innych czynności — to naprowadzi Cię na właściwe kryterium zachłannego postępowania.

Problem 13.34. Dokładne rozwiązanie problemu reszty, zdefiniowanego w zadaniu 11.4, można znaleźć metodą programowania dynamicznego.

(A) Zmodyfikuj zależności (11.1), (11.2) i (11.3) tak, aby stanowiły matematyczny zapis problemu wydawania reszty za pomocą najmniejszej liczby monet.

(B) Wyprowadź algorytm programowania dynamicznego dla problemu reszty.

Uwaga. W punkcie (A) musisz uwzględnić, że liczba monet w reszcie jest minimalizowana i reszta ma być dokładnie wydana.

Problem 13.35. W punkcie 1.2 sformułowaliśmy **problem komiwojażera**, który polega na znalezieniu najkrótszej drogi zamkniętej, przechodzącej przez każdy punkt (miejscowość) dokładnie jeden raz. Za długość drogi przyjmujemy jak zwykle sumę długości jej odcinków między kolejnymi punktami. Z informacji tam podanych wynika, że bardzo trudno jest znaleźć optymalne (czyli najkrótsze) rozwiązanie tego problemu. Tutaj proponujemy zastanowić się nad znalezieniem **rozwiązania przybliżonego**, tzn. takiego, że nie potrafimy wykazać, iż jest ono najkrótszą drogą komiwojażera, ale które z pewnych względów możemy uznać za lepsze niż dowolną drogę zamkniętą przechodzącą przez każdy punkt dokładnie jeden raz.

(A) Opisz algorytm znajdowania drogi komiwojażera, który można nazwać metodą **najbliższego sąsiada**, a jest realizacją następującej strategii zachłannej: poczynawszy od punktu startu, jako następny punkt wybierz ten, w którym jeszcze nie byłeś, a który leży najbliżej punktu, w którym się znajdujesz. Z ostatniego punktu wróć do punktu startu. Zaczynając od Warszawy, zastosuj ten algorytm do znalezienia drogi zamkniętej przechodzącej przez wszystkie miasta wojewódzkie z podziału administracyjnego kraju sprzed 1997 roku (odległości między tymi miastami odczytaj z atlasu drogowego lub kolejowego). Porównaj to rozwiązanie z trasą pokazaną na rysunku 1.1. Zastosuj ten algorytm również do podziału administracyjnego z lat 1975 – 1999, gdy Polska była podzielona na 49 województw. Odpowiednie dane do obu podziałów administracyjnych Polski znajdziesz w internecie.

(B) Inny rodzaj rozwiązań problemu komiwojażera otrzymuje się **metodami włączania**. Przypuśćmy, że znana jest trasa zamknięta przechodząca przez pewien zbiór punktów i masz dołączyć do niej kilka nowych punktów. W jakiej kolejności będziesz je włączał i w które miejsca trasy, aby otrzymać możliwie najkrótszą drogę zamkniętą? Zaczynając budowanie od dowolnego punktu, możesz w ten sposób otrzymać trasę przechodzącą przez wszystkie punkty. Zaproponuj odpowiedni algorytm. Ponownie, zastosuj ten algorytm do przykładów z poprzedniego punktu.

(C) Naszkicowaną w punkcie (B) metodę włączania zastosuj do otrzymania trasy komiwojażera przechodzącej przez wszystkie miasta wojewódzkie z podziału administracyjnego z lat 1997 – 1999. W tym celu, jako rozwiązanie częściowe, przyjmij najkrótszą trasę przechodzącą przez miasta wojewódzkie z podziału administracyjnego sprzed 1975 roku pokazaną na rysunku 1.1.

Uwagi. Problem komiwojażera ma wiele zastosowań praktycznych. Wyobraźmy sobie maszynę, która nituje pokrycia skrzydeł samolotu. Porusza się ona od punktu do punktu i byłoby dobrze, gdyby swoje zadanie mogła wykonać w najkrótszym czasie, czyli przebywając najkrótszą drogę między wszystkimi punktami nitowania. A takich punktów na jednej połączy skrzydła jest od kilku do kilkudziesięciu tysięcy! Istnieją problemy o jeszcze większych rozmiarach.

Maszyna, która produkuje układy scalone, popularnie zwane kośćmi, musi napylić pewną substancję w miejscu, gdzie ma się znaleźć tranzystor. I podobnie, chcemy zaplanować dla niej najkrótszą drogę przez wszystkie miejsca przeznaczone na tranzystory, których w układach VLSI (tj. w układach o bardzo dużej skali integracji) może być kilkaset tysięcy!

Zagadnienia praktyczne o tak olbrzymich rozmiarach muszą być rozwiązywane bardzo szybkimi metodami. Gdy nie są znane efektywne algorytmy dokładne, dużego znaczenia nabierają szybkie metody znajdowania rozwiązań przybliżonych.

Z algorytmami znajdowania dokładnych i przybliżonych rozwiązań problemu komiwojażera możesz się zapoznać w książce [Sysło].

Rozdział 14. Gdzie szukać dalszych informacji o algorytmach

Tu dowiesz się:

- ▶ w jakich opracowaniach szukać dalszych informacji o algorytmach;
- ▶ w skrócie, jakim aspektem algorytmiki są poświęcone te opracowania.

14.1. Opracowania podstawowe

[EI] Elementy informatyki. Pakiet oprogramowania edukacyjnego, Instytut Informatyki Uniwersytetu Wrocławskiego, OFEK, Wrocław, Poznań 1993.

Jest to dokumentacja dużego pakietu oprogramowania edukacyjnego, znanego pod nazwą pakiet EI. Składa się on z 10 systemów i jest przeznaczony do wspomagania nauczania elementów informatyki. Wyprodukowano i przekazano szkołom blisko 2000 egzemplarzy tego pakietu. Dokumentacja zawiera opis sposobów posługiwania się tym oprogramowaniem oraz ćwiczenia i wskazówki metodyczne, pożyteczne podczas zajęć. Zwięzły opis możliwości pakietu EI przedstawiono w artykule zamieszczonym w kwartalniku „Komputer w Edukacji”, nr 1/1994.

Pakiet EI został utworzony w środowisku systemu operacyjnego DOS, może być również uruchamiany pod nadzorem systemu operacyjnego Windows. Ze względu jednak na olbrzymi wzrost szybkości działania procesorów, praca z tym pakietem w trybie demonstracyjnym jest obecnie znacznie utrudniona i nie daje pożądanych efektów. Niektóre programy z tego pakietu zostały na nowo zrealizowane w środowisku systemu Windows, wśród nich porządkowanie. Pojawiły się także nowe programy edukacyjne, przeznaczone do demonstracji działania wybranych grup algorytmów. Wiele z tych programów jest dostępnych na stronie [Oprog].

[EI-I] Elementy informatyki. Podręcznik pod redakcją Macieja M. Sysły, Wyd. 8, Wydawnictwo Naukowe PWN, Warszawa 1997.

[EI-II] Elementy informatyki. Rozwiązania zadań pod redakcją Macieja M. Sysły, Wyd. 3, Wydawnictwo Naukowe PWN, Warszawa 1995.

[EI-III] Elementy informatyki. Poradnik metodyczny dla nauczyciela pod redakcją Macieja M. Sysły, Wydawnictwo Naukowe PWN, Warszawa 1997.

Duży nacisk w tych opracowaniach, dostępnych w wielu bibliotekach i nadal wykorzystywanych przez wielu nauczycieli, położono na metody rozwiązywania problemów i algorytmikę. Rozwiązania wszystkich problemów zawierają ich

pełne komputerowe realizacje w wybranym języku programowania (Logo lub Pascal) lub w systemie użytkowym (edytor tekstów, arkusz kalkulacyjny, baza danych).

[ELI] ELBOX. Laboratorium informatyki ELI. Instrukcja użytkownika, Warszawa 1996.

Program ELI był powszechnie wykorzystywany w wielu książkach i podręcznikach do informatyki, publikowanych w latach 1995 – 2005, w szczególności był podstawowym programem edukacyjnym, użytym przez autora w tej książce wydawanej wcześniej przez WSiP. Ten program nie jest jednak dalej rozwijany i nie można go uruchomić na komputerach z nowszymi systemami operacyjnymi. To było m.in. powodem rezygnacji z tego programu w tym wydaniu tej książki.

[Harel] Dawid Harel, Algorytmika. Rzecz o istocie informatyki, WNT, Warszawa 1992.

Popularny i w miarę elementarny, przynajmniej na początku, wykład na temat algorytmów, metod ich konstruowania oraz ich własności. Jest to dobre wprowadzenie do informatyki.

[Inf-I] Ewa Gurbiel, Grażyna Hardt-Olejniczak, Ewa Kołczyk, Helena Krupicka, Maciej M. Sysło, Informatyka. Część 1. Podręcznik dla liceum ogólnokształcącego. Kształcenie w zakresie rozszerzonym, Wyd. 2, WSiP, Warszawa 2003.

[Inf-II] Ewa Gurbiel, Grażyna Hardt-Olejniczak, Ewa Kołczyk, Helena Krupicka, Maciej M. Sysło, Informatyka. Część 2. Podręcznik dla liceum ogólnokształcącego. Kształcenie w zakresie rozszerzonym, Wyd. 2, WSiP, Warszawa 2004.

[Inf-III] Ewa Gurbiel, Grażyna Hardt-Olejniczak, Ewa Kołczyk, Helena Krupicka, Maciej M. Sysło, Informatyka. Poradnik dla nauczycieli i program nauczania w liceum ogólnokształcącym. Kształcenie w zakresie rozszerzonym, WSiP, Warszawa 2004.

Duży nacisk w tych trzech publikacjach położono na metody rozwiązywania problemów i algorytmikę. Rozwiązania większości problemów zawierają ich pełne komputerowe realizacje w wybranym języku programowania (Pascal lub Delphi) lub w systemie użytkowym (edytor tekstów, arkusz kalkulacyjny, baza danych, system do tworzenia stron WWW). Materiały elektroniczne zamieszczono na dołączonych do podręczników płytach. W treść podręczników wpleciono prezentację i omówienie zadań maturalnych. Książki te będą dostępne w serwisie internetowym, towarzyszącym tej książce:

<http://edukacja.helion.pl/algorytmika>.

[Log] Maciej M. Sysło, Anna Beta Kwiatkowska, Myśl logarytmicznie!, Delta

[OI] Olimpiada informatyczna, (materiały wydawane co roku) Komitet Główny Olimpiady Informatycznej.

Materiały z kolejnych olimpiad informatycznych, zwane „niebieskimi książeczkami”, zawierają m.in. teksty wszystkich zadań konkursowych wraz z rozwiązaniami. Polecamy te opracowania zwłaszcza tym, którzy interesują się bardziej złożonymi problemami (takimi, jak w punkcie 13.2) oraz informatycznymi metodami ich rozwiązywania. Można je pobrać z oficjalnej strony Olimpiady <http://www.oi.edu.pl/>.

[Oprog] Na stronie <http://mmsyslo.pl/Materialy/Oprogramowanie> jest udostępnionych wiele programów edukacyjnych, których celem jest przybliżenie uczącym się metod algorytmicznych i ich realizacji w różnych systemach.

[Piramidy] Maciej M. Sysło, Piramidy, szyszki i inne konstrukcje algorytmiczne, Helion, Gliwice 2015.

Książka jest wyborem rzeczywistych sytuacji problemowych, na których przykładzie zilustrowano różne sposoby otrzymywania ich rozwiązań w postaci algorytmicznej i w wielu fragmentach stanowi poszerzenie rozważań prowadzonych w tej książce (zaznaczamy to w wielu miejscach) Autor podaje pod rozwagę: czy przepisy kulinarne są algorytmami i jak budowano piramidy, w jaki sposób dobierać się w trwałe pary oraz wydawać resztę najmniejszą liczbą monet. Zastanawia się również, skąd bierze się taka powszechność występowania liczb Fibonacciego w przyrodzie i jak powinny być zbudowane automaty do kawy, byśmy nie czekali na nią zbyt długo. Teksty programów służących do rozwiązywania wybranych problemów z tej książki są dostępne na stronie <http://edukacja.helion.pl/algorytmika>.

[Plansze Historia] Maciej M. Sysło, Historia informatyki. Plansze do pracowni komputerowej, WSiP, Warszawa 2006.

George Pólya (1887–1985) — światowej sławy matematyk pochodzenia węgierskiego, który zasłynął z umiejętności przystępnego opisywania rozwiązań często trudnych problemów.

Zestaw 12 dwustronnych plansz w formacie A1, obejmujących najważniejsze fakty historyczne związane z rachowaniem, przyrządami do rachowania i algorytmami, które doprowadziły w czasach obecnych do powstania komputerów i informatyki, a następnie rozwoju współczesnej komunikacji elektronicznej. To bogato ilustrowana, syntetyczna historia informatyki i komputerów od czasów najdawniejszych do ery Internetu. Obecnie te plansze są dostępne w wersji elektronicznej na stronie <http://mmsyslo.pl/Historia/Plansze-z-historii-informatyki>.

[Pólya] George Pólya, Jak to rozwiązać?, Wyd. 2, Wydawnictwo Naukowe PWN,

Warszawa 1993.

Jest to klasyczna już książeczka o rozwiązywaniu zadań i problemów matematycznych. Autor objaśnia w niej sposoby znajdowania rozwiązań, przedstawia je oraz podaje zasady, które odnoszą się również do rozwiązywania problemów z innych dziedzin. Pisze m.in. tak: *Rozwiązywanie zadań jest taką samą umiejętnością praktyczną, jak np. pływanie. Każdą praktyczną umiejętność osiągamy przez naśladowanie i ćwiczenie. I do tego zachęcamy również w naszej książce o algorytmach. Zasady rozwiązywania problemów i tworzenia algorytmów są bardzo podobne, m.in. namawiamy do: ścisłego opisywania rozwiązywanych problemów, przestrzegania kolejnych etapów rozwiązywania (zrozum problem — czyli sformułuj jego specyfikację, ułóż plan rozwiązywania — tzn. algorytm, wykonaj plan — czyli podaj realizację algorytmu, sprawdź otrzymane rozwiązanie — czyli sprawdź poprawność algorytmu), korzystania z analogii i z rozwiązań zadań pokrewnych. G. Pólya przybliżyła w swoich rozważaniach znaczenie heurystyki — dziedziny badawczej, zajmującej się wykrywaniem prawidłowości twórczego myślenia i działania; tym terminem określa się również naturalne metody znajdowania rozwiązań problemów, metody heurystyczne — są nimi na przykład algorytmy „z ręką na ścianie” oraz algorytmy zachłanne (rozdział 11.).*

[Steinhaus] Hugo Steinhaus Kalejdoskop matematyczny Wyd. 4 zmienione, WSiP, Warszawa 1989.

Hugo Dionizy Steinhaus (1887-1972) — polski matematyk, współtwórca lwowskiej szkoły matematycznej, a później założyciel wrocławskiej szkoły zastosowań matematyki. Znany również jako krzewiciel języka polskiego i autor błyskotliwych aforyzmów i powiedzeń, nie tylko związanych z matematyką. Przykłady powiedzeń o matematyce (zaczepnięte ze *Słownika racjonalnego* Ossolineum, Wrocław 1993):

* *Między duchem a materią pośredniczy matematyka.*

* *Łatwo z domu rzeczywistości zejść do lasu matematyki, ale nieliczni tylko umieją wrócić.*

* *Matematyka nie może wypełnić życia, ale nieznanomość matematyki już niejednemu wypełniła.*

* *Pan A. unika podręczników algebry, odkąd znalazł tam twierdzenie $A = A$.*

Pierwsze wydanie tej książki ukazało się w 1938 roku we Lwowie, jednocześnie z wydaniem angielskim. W następnych latach przetłumaczono ją na wiele języków. Hugo Steinhaus zawarł w niej wiele problemów i ich rozwiązań, które obecnie należą do trwałych zasobów algorytmiki — wielokrotnie wspominamy o nich podczas naszych rozważań. Bardzo zachęcamy do lektury tej książki, w której rozwiązania trudnych problemów matematycznych są wyjaśnione na przykładach rzeczywistych sytuacji i obiektów, często z pomocą odpowiednio

dobrych ilustracji.

[Wilson] Robin J. Wilson, Wprowadzenie do teorii grafów, PWN, Warszawa 1985.

W punkcie 11.1.3 omawiamy problem znajdowania wyjścia z labiryntu i podajemy algorytm wyznaczania najkrótszej takiej drogi, który jest szczególnym przypadkiem algorytmu Dijkstry znajdowania najkrótszej drogi w grafie. Grafy, zwłaszcza ich szczególne rodzaje — drzewa, pojawiają się w naszej książce wielokrotnie. Książka R. J. Wilsona jest wprowadzeniem do ogólniejszych rozważań na temat grafów, które są podstawowymi obiektami rozważań w algorytmice.

Niklaus Wirth jest twórcą takich języków programowania, jak: Pascal i Modula. Konstrukcje programistyczne w tych językach ściśle odpowiadają podstawowym krokom większości algorytmów i z tego m.in. względu te języki nazywa się często językami algorytmicznymi.

[Wirth] Niklaus Wirth, Algorytmy + Struktury Danych = Programy, Wyd. 2, WNT, Warszawa 1980.

Książka jest poświęcona realizacjom podstawowych algorytmów w języku Pascal. Rozważania są znakomitą ilustracją tezy zawartej w jej tytule, mówiącej że program komputerowy działa na podstawie algorytmu, który z kolei jest ściśle związany ze strukturami danych, na których operuje.

14.2. Opracowania zaawansowane

[BanKrecz] Lech Banachowski, Antoni Kreczmar, Elementy analizy algorytmów, WNT, Warszawa 1982.

Cechą odróżniającą tę książkę od pozostałych są formalne dowody poprawności przedstawionych w niej algorytmów.

[Bentley] Jon Bentley, Perełki oprogramowania, WNT, Warszawa 1986.

Ta publikacja jest poświęcona realizacjom algorytmów, których omówienie jest posunięte o krok dalej niż w naszej książce — są to bowiem głównie komputerowe realizacje algorytmów, uwzględniające postać i wielkość rzeczywistych danych, pochodzących często z praktycznych zastosowań. Polecamy ją przede wszystkim tym, których naprawdę interesują perełki oprogramowania, czyli dość wyszukane sposoby komputerowych realizacji algorytmów.

[Cormen] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Wprowadzenie do algorytmów, Wyd. 8, WNT, Warszawa 2007.

Książka jest wprowadzeniem do metod analizy algorytmów. Od połowy lat 90.

XX wieku jest to najpopularniejsze opracowanie dotyczące algorytmiki na poziomie akademickim.

[Graham] Ronald L. Graham, Donald E. Knuth, Oren Patashnik, Matematyka konkretna, Wydawnictwo Naukowe PWN, Warszawa 1996.

Książka zawiera podstawowe wiadomości dotyczące matematycznych własności liczb, działań i funkcji, które występują w algorytmach zarówno na etapie ich konstruowania, jak i analizowania.

[Knuth-1] Donald E. Knuth, Sztuka Programowania, tom 1, Algorytmy podstawowe, WNT, Warszawa 2003.

[Knuth-2] Donald E. Knuth, Sztuka Programowania, tom 2, Algorytmy seiniuneryczne, WNT, Warszawa 2003.

[Knuth-3] Donald E. Knuth, Sztuka Programowania, tom 3, Sortowanie i wyszukiwanie, WNT, Warszawa 2003.

Sztuka programowania — to tłumaczenie pierwszej części tytułu tego wielotomowego dzieła, które jest poświęcone przede wszystkim algorytmom. W dziedzinach, których dotyczą te opracowania, autor przedstawia *state of the art* — wyczerpujące omówienie tematów. Można w nich znaleźć niemal każdy z omówionych przez nas problemów, potraktowany dogłębnie, w pełnej ogólności i z najdrobniejszymi szczegółami dotyczącymi działania i własności rozwiązań, czyli algorytmów.

[Sysło] Maciej M. Sysło, Narsingh Deo, Janusz S. Kowalik, Algorytmy optymalizacji dyskretniej z programami w języku Pascal, Wyd. 2, Wydawnictwo Naukowe PWN, Warszawa 1995.

Książka ta stanowi pomost między matematycznym, ścisłym opisem algorytmów a ich komputerowymi realizacjami. Na przykładach klasycznych problemów z matematyki dyskretniej, takich jak: programowanie liniowe i całkowitoliczbowe, problemy pokrycia (m.in. problem plecakowy), problemy optymalizacyjne na grafach (m.in. problemy najkrótszych dróg) oraz problemy szeregowania zadań zilustrowano sposoby doboru struktur danych oraz tworzenia programów komputerowych rozwiązujących te problemy. Realizacje 28 klasycznych algorytmów optymalizacyjnych w języku Pascal są dostępne na stronie <http://edukacja.helion.pl/algorytmika> towarzyszącej tej książce. Książka ta jest obecnie dostępna w wersji elektronicznej na stronie Wydawnictwa Naukowego PWN.

Wskazanie innych opracowań poświęconych algorytmice ma Cię zachęcić do dalszej lektury. W tej książce dotknęliśmy tylko czubka góry lodowej. Umiejętność rozwiązywania problemów, na którą składa się tworzenie algorytmów, jak każdą umiejętność umysłową, kształcić trzeba nieustannie.

Rozdział 15. Algorytmika w zadaniach maturalnych

W tym rozdziale zamieszczamy przykładowe zadania maturalne z informatyki^[1], które ze względu na swój zakres można zaliczyć do algorytmiki.

W przypadkach niektórych zadań, blisko związanych z zagadnieniami omawianymi w tej książce, odsyłamy do odpowiednich jej fragmentów. Wiele innych zadań maturalnych jest wplecionych w treść podręczników [Inf-I] i [Inf-II] i szerzej tam skomentowanych. Zadania są zamieszczone poniżej w kolejności chronologicznej. Komentarze przy końcu zadań zostały dopisane na potrzeby tej książki i nie były częścią treści zadań maturalnych.

Zadanie Potęgowanie

(Informator maturalny od 2005 z informatyki, Arkusz I, CKE, Warszawa 2003)

Poniżej są podane dwa sposoby obliczania wartości potęg liczb o wykładnikach naturalnych. Pierwszy sposób jest opisany za pomocą definicji indukcyjnej, a drugi — za pomocą algorytmu zapisanego w postaci listy kroków.

Sposób I:

$$a^0 = 1 \text{ dla } a \in \mathbb{R} \setminus \{0\},$$

$$a^n = a^{n-1} \cdot a \text{ dla } n \in \mathbb{N}^+, a \in \mathbb{R} \setminus \{0\},$$

Sposób II:

Specyfikacja problemu

Dane: a — podstawa potęgi, n — wykładnik potęgi dla $n \in \mathbb{N}^+, a \in \mathbb{R} \setminus \{0\}$.

Wyniki: *wynik* — wartość potęgi o podstawie a i wykładniku n , $\text{wynik} \in \mathbb{R}$.

Zmienne pomocnicze: x, k .

Krok 1. Nadaj wartości zmiennym: zmiennej *wynik* wartość 1, zmiennej x wartość a , zmiennej k wartość n .

Krok 2. Dopóki $k \neq 0$, powtarzaj krok 3.

Krok 3. Jeśli k jest liczbą nieparzystą, to *wynik* pomnóż przez x , zaś k zmniejsz o 1, w przeciwnym razie k podziel przez 2, zaś x pomnóż przez x .

Krok 4. Wypisz wartość *wynik*.

Wykonaj polecenia:

a) Zapisz rekurencyjną funkcję obliczania potęgi a^n w wybranym przez siebie języku (pseud języku) programowania.

- b)** Utwórz schemat blokowy algorytmu opisanego jako sposób II.
- c)** Załóżmy, że mamy obliczyć wartość 15^{1000} . Którego sposobu należy użyć? Przed podjęciem decyzji wyznacz złożoność obliczeniową (czasową) i opisz złożoność pamięciową obu wymienionych sposobów. Krótko uzasadnij swój wybór.

KOMENTARZ. Zauważmy, że Sposób I obliczania wartości potęgi jest algorytmem rekurencyjnym, odwołującym się tylko do poprzedniej wartości wykładnika. Sposób II zaś to algorytm rekurencyjny, w którym odwołania są do wykładników prawie o połowę mniejszych, porównaj punkt 8.1.2. To znacznie przyspiesza obliczenia, o czym można się przekonać, wykonując część c) zadania. Warto zauważyć, że kolejność mnożeń przy obliczaniu potęgi wynikająca ze Sposobu II jest taka sama, jak w algorytmie, który wynika z rozkładu wykładnika na postać binarną i zastosowania schematu Hornera do tej postaci, porównaj punkt 7.3.2.

Zadanie Rozmnażanie się pszczoł

(Informator maturalny od 2005 z informatyki, Arkusz I, CKE, Warszawa 2003)

Pszczoły rozmnażają się tak, że z zapłodnionych jaj rodzą się samice, a z niezapłodnionych samce (trutnie). Rodzina trutnia jest nietypowa: brak ojca, tylko jeden dziadek i jedna babcia, jeden pradziadek, ale dwie prababcie itd.

Uwaga: Rozwiązując zadania, przyjmij, że pokolenie 0 to pokolenie rodziców, 1 to pokolenie dziadków, 2 — pradziadków itd.

- a)** Narysuj drzewo genealogiczne trutnia do piątego pokolenia wstecz włącznie.
- b)** Zapisz rekurencyjny wzór ciągu, który pozwala obliczyć liczbę męskich przodków w n -tym pokoleniu.
- c)** Oblicz, ilu męskich przodków ma truteń w piątym i dziesiątym pokoleniu. Zapisz obliczenia.
- d)** Poniżej jest podany schemat blokowy algorytmu służącego do obliczania liczby męskich przodków trutnia w n -tym pokoleniu wstecz w sposób iteracyjny. Schemat ten zawiera luki. Uzupełnij puste miejsca odpowiednimi instrukcjami i warunkami z listy zamieszczonej po schemacie. Zwróć uwagę na odpowiednią kolejność wpisywanych instrukcji. Uzupełnij również opisy użytych zmiennych.

Specyfikacja problemu

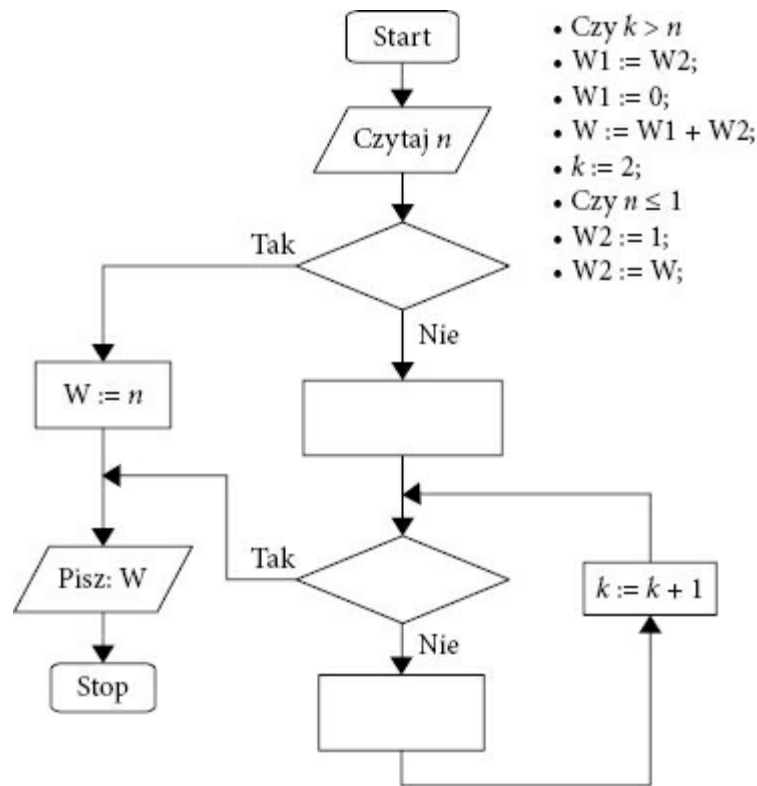
Dane wejściowe $n \in \mathbb{N}$

Wynik $W \in \mathbb{N}^+$

Nazwa zmiennej Opis zmiennej

W1, W2

Schemat blokowy z blokami częściowo pustymi, a częściowo wypełnionymi poniższymi napisami:



KOMENTARZ. Po narysowaniu przykładowego drzewa genealogicznego trutnia (punkt a) stanie się jasne, że ich liczby w poszczególnych pokoleniach mają coś wspólnego z... królikami Fibonacciego (porównaj punkt 8.2.2). Udzielenie odpowiedzi na pozostałe punkty zadania staje się już łatwe. *Uwaga.* W punkcie d) w niektóre bloki schematu należy wpisać więcej niż jedną instrukcję.

Zadanie Szeregi nieskończone i funkcje elementarne

(Egzamin maturalny z informatyki, Arkusz I, 2005)

Wartości funkcji elementarnych, takich jak sin, cos, log, są obliczane za pomocą komputera w sposób przybliżony. Często stosuje się w tym celu wzory, które mają postać nieskończonych sum. Na przykład prawdziwy jest następujący wzór na wartość logarytmu naturalnego, czyli przy podstawie e , z liczby 2:

$$\ln 2 = \frac{2}{3} \left(1 + \frac{1}{3} \cdot \frac{1}{9} + \frac{1}{5} \cdot \frac{1}{9^2} + \frac{1}{7} \cdot \frac{1}{9^3} + \frac{1}{9} \cdot \frac{1}{9^4} + \dots \right)$$

W oparciu o powyższy wzór można zaprojektować i napisać program, który dla danej liczby ε ($\varepsilon > 0$) oblicza przybliżoną wartość $\ln 2$, sumując jak najmniej wyrazów, aby różnica między dwoma ostatnimi przybliżeniami była mniejsza niż ε .

Wprowadźmy oznaczenie:

dla $n \geq 1$

$$l_n = \frac{2}{3} \left(1 + \frac{1}{3} \cdot \frac{1}{9} + \frac{1}{5} \cdot \frac{1}{9^2} + \frac{1}{7} \cdot \frac{1}{9^3} + \dots + \frac{1}{2n+1} \cdot \frac{1}{9^n} \right)$$

$$l_0 = 2/3$$

a) Wypełnij tabelę:

$N \ln$

0

1

2

3

Podaj zależność pomiędzy wartościami \ln i $\ln - 1$ dla każdego $n = 1, 2, \dots$

Podaj wzór rekurencyjny na różnicę $r_n = \ln - \ln - 1$ dla $n > 0$.

b) Podaj algorytm ze specyfikacją (w postaci listy kroków, schematu blokowego lub w języku programowania), który dla danej liczby ε ($\varepsilon > 0$) oblicza przybliżoną wartość $\ln 2$, sumując jak najmniej wyrazów we wzorze podanym w treści zadania, aby różnica między dwoma ostatnimi przybliżeniami była mniejsza niż ε .

KOMENTARZ. Algorytm, który należy podać w punkcie c), jest podobny do algorytmu iteracyjnego służącego do obliczania przybliżonej wartości pierwiastka kwadratowego (porównaj punkt 7.7) — we wzorze na $\ln 2$, w nawiasie, należy dodać kolejny składnik, jeśli kolejna różnica r_n nie jest mniejsza od ε . *Uwaga.* Do rozwiązania tego zadania nie trzeba wiedzieć, ani co to jest logarytm naturalny, ani w jaki sposób otrzymano podany wzór na wartość $\ln 2$.

Zadanie Ewolucja

(Egzamin maturalny z informatyki, Arkusz I, 2005)

Na planecie MLAP każdy żyjący organizm ma postać napisu złożonego z wielkich liter alfabetu łacińskiego. Każdy nowo powstały organizm jest opisywany literą A. Po każdym roku życia wielkość organizmu podwaja się w taki sposób, że każda z liter zostaje zastąpiona dwiema literami zgodnie z pewnym ustalonym zbiorem reguł postaci:

$L \rightarrow F S$,

oznaczających, że literę L można zastąpić przez dwie litery: $F S$. O literze L mówimy wówczas, że występuje po lewej stronie reguły, a F i S występują po prawej stronie reguły. Przez wielkość organizmu rozumiemy tutaj długość odpowiedniego napisu. Rozważmy następujący zbiór reguł:

$A \rightarrow B C \quad B \rightarrow A D \quad D \rightarrow A A$

$A \rightarrow C D C \rightarrow B A D \rightarrow B B$

Wówczas organizmy roczne mogą przyjąć jedną z postaci: $B C$, $C D$, zaś dwuletnie

$A D B A$ ($A \rightarrow B C \rightarrow A D B A$) $B A B B$ ($A \rightarrow C D \rightarrow B A B B$)

$B A A A$ ($A \rightarrow C D \rightarrow B A A A$)

O dwóch organizmach mówimy, że są w danym momencie odróżnialne, jeśli różne są odpowiadające im napisy (mają różne długości lub różnią się na co najmniej jednej pozycji).

a) Wypisz poniżej wszystkie odróżnialne organizmy trzyletnie, które można uzyskać z organizmu dwuletniego o postaci $A D B A$.

b) Podaj sposób sprawdzania dla danej liczby naturalnej $n > 1$, czy mogą istnieć organizmy o długości n . W przypadku odpowiedzi pozytywnej należy również ustalić wiek organizmu o wielkości n . Podaj, ile poprawnych wielkości organizmów występuje w przedziale $(n, m]$ dla liczb naturalnych n i m , gdzie $n < m$. Odpowiedź uzasadnij.

c) Przyjmijmy, że każda litera, pojawiająca się w regułach, występuje dokładnie raz po lewej stronie reguły, przed „strzałką” (zauważmy, że powyższy przykład nie spełnia tego warunku, ponieważ litery A i D występują każda z lewej strony w dwóch regułach). Ile odróżnialnych organizmów w wieku 1, 2, 3 itd. może wówczas występować? Odpowiedź uzasadnij.

d) Poniżej przedstawiona jest funkcja wspomagająca realizację następującego zadania: dla zadanego zbioru reguł, nowo powstałego organizmu *start* i danego napisu należy ustalić, czy napis ten przedstawia organizm, który można uzyskać przy pomocy reguł zadanych w treści zadania.

Niech: $L_1 \rightarrow F_1 S_1, L_2 \rightarrow F_2 S_2, \dots, L_p \rightarrow F_p S_p$ — dany zbiór reguł.

Specyfikacja funkcji sprawdź

Dane: *napis* — ,

start — ,

Wynik: odpowiedź, czy napis przedstawia organizm, który można uzyskać przy pomocy podanych reguł, gdy nowo powstały organizm jest opisywany przez *start*.

Treść funkcji sprawdź:

jeśli długość napisu nie jest potęgą liczby 2, to zakończ wykonywanie funkcji z odpowiedzią NIE

w przeciwnym razie wykonuj:

▶ jeśli *napis* = *start*, to zakończ wykonywanie funkcji z odpowiedzią TAK;

▶ jeśli długość napisu jest równa 1, to zakończ wykonywanie funkcji z odpowiedzią NIE;

▶ podziel napis na dwie równe części: *napis1* i *napis2*;

▶ dla $i = 1, 2, \dots, p$ wykonuj:

jeśli $L_i = \textit{start}$, to:

■ wykonaj funkcję *sprawdź* rekurencyjnie dla $\textit{napis} = \textit{napis1}$, $\textit{start} = F_i$ oraz dla $\textit{napis} = \textit{napis2}$ i $\textit{start} = S_i$;

■ jeśli oba rekurencyjne wywołania funkcji *sprawdź* zakończyły się odpowiedzią TAK, to zakończ wykonywanie funkcji z odpowiedzią TAK;

▶ jeśli w powyższej pętli nie zakończyliśmy działania funkcji, to zakończ jej wykonywanie z odpowiedzią NIE.

Uzupełnij specyfikację podanej powyżej funkcji.

Podaj parametry wszystkich rekurencyjnych wywołań funkcji *sprawdź* przy uruchomieniu jej dla następującego zbioru reguł:

$A \rightarrow B \quad C B \rightarrow A \quad D D \rightarrow A \quad A$

$A \rightarrow C \quad D C \rightarrow B \quad A D \rightarrow B \quad B$

oraz $\textit{napis} = B C A A A D C D$ i $\textit{start} = A$.

Jaką odpowiedź da funkcja w tym przypadku?

KOMENTARZ. Odpowiedź na pierwsze pytanie w punkcie b) jest zawarta... w treści funkcji *sprawdź*. Dalsza część odpowiedzi w punkcie b) wymaga posłużenia się funkcją logarytm lub dzieleniem przez 2 (funkcja logarytm i dzielenie są działaniami odwrotnymi do potęgowania i mnożenia). Funkcja *sprawdź* jest rekurencyjną realizacją działania odwrotnego do tworzenia organizmów.

Zadanie Najlepsze sumy, najpopularniejsze elementy

(Informator 2005, Arkusz II, 2005)

Najlepszą sumą ciągu liczb a_1, a_2, \dots, a_n nazywamy największą wartość wśród sum złożonych z **sąsiednich** elementów tego ciągu. Na przykład dla ciągu: 1, 2, -5, 7 mamy następujące sumy:

1, $1 + 2 = 3$, $1 + 2 + (-5) = -2$, $1 + 2 + (-5) + 7 = 5$, **2**, $2 + (-5) = -3$, $2 + (-5) + 7 = 4$, **-5**, $-5 + 7 = 2$, **7**.

Zatem najlepszą sumą jest 7 (zwróć uwagę, że jeden element też uznajemy za sumę).

Do oceny oddajesz:

Na nośniku WYNIKI dokument tekstowy *Raport5* zawierający odpowiedzi do punktów a), b), c).

Wykonaj poniższe polecenia.

a) Dany jest następujący ciąg liczb całkowitych: 1, -2, 6, -5, 7, -3. Wyznacz najlepszą sumę dla tego ciągu i wpisz jej wartość **Najlepsza suma**.

Czy na podstawie uzyskanego wyniku można podać wartość najlepszej sumy dla ciągu:

1, -2, 2, 2, 2, -5, 3, 3, 1, -3.

Do oceny oddajesz w dokumencie *Raport5* wartości najlepszej sumy dla ciągu oraz odpowiedź z uzasadnieniem na powyższe pytanie.

b) Zaproponuj algorytm wyznaczania najlepszej sumy dla dowolnego ciągu liczb całkowitych. Na jego podstawie napisz program do obliczenia najlepszych sum ciągów liczb podanych w plikach *dane5-1.txt*, *dane5-2.txt*, *dane5-3.txt*.

Do oceny oddajesz także w dokumencie *Raport5*:

► opis algorytmu zawierającego odpowiednie fragmenty kodu Twojego programu,

► wartości najlepszych sum dla poszczególnych plików, które wpisałeś do powyższej tabeli.

c) Wyznacz „najpopularniejszy” element w ciągu, czyli element występujący największą liczbę razy. Zaprojektuj jak najszybszy algorytm wyznaczania najpopularniejszego elementu ciągu oraz oszacuj liczbę wykonywanych przez niego operacji (czas działania) jako funkcję od liczby elementów w ciągu. Zaprogramuj swój algorytm i zastosuj go do ciągów znajdujących się w plikach *dane5-1.txt*, *dane5-2.txt*, *dane5-3.txt*. W przypadku, gdy w ciągu jest więcej niż jeden najpopularniejszy element, jako wynik podajemy dowolny z nich. Na przykład dla ciągu 1, 3, 5, 1, 3 poprawną odpowiedzią jest zarówno 1, jak i 3 (oba elementy występują dwa razy).

Do oceny oddajesz w dokumencie *Raport5*:

► najpopularniejsze elementy w plikach *dane5-1.txt*, *dane5-2.txt*, *dane5-3.txt* umieszczone w tabeli czytelnie prezentującej te wyniki,

► opis algorytmu zawierającego odpowiednie fragmenty kodu Twojego programu oraz oszacowanie czasu jego działania.

KOMENTARZ. *Uwaga.* W rozwiązaniu punktu c) zadania nie korzysta się z rozwiązania części a) i b). Rozwiązanie części a) i b) jest przedyskutowane w problemie 13.23, gdzie podano również nietrywialny algorytm i jego kod.

Punkt c) zadania można rozwiązać przynajmniej na trzy sposoby (zakładamy, że ciąg danych zawiera n liczb):

Algorytm 1. Bez względu na charakter danych, najpierw porządkujemy dany ciąg, a następnie — przeglądając go od lewej do prawej — zliczamy, ile jest takich samych elementów i których jest najwięcej. Złożoność tego algorytmu wynosi około $n \log n + n$, pierwszy składnik to złożoność porządkowania n liczb (porównaj rozdział 10.), a drugi — to złożoność przejrzenia wszystkich elementów ciągu.

Algorytm 2. Najpierw sprawdzamy, jaki charakter mają dane ciągi. Okazuje się, że nawet w najdłuższym z nich jest niewiele elementów różnych. W takim przypadku można skorzystać z algorytmu kubełkowego, porównaj punkt 6.3.1. Złożoność tego algorytmu jest proporcjonalna do n , liczby elementów ciągu, pod warunkiem, że dane ciągi zawierają niewiele elementów różnych.

Algorytm 3. Podobnie jak w przypadku Algorytmu 2., ten sposób rozwiązania zależy również od wartości elementów w danych ciągach. W tym algorytmie można posłużyć się arkuszem kalkulacyjnym Excel i jego predefiniowaną funkcją CZĘSTOŚĆ. Szczegóły pozostawiamy do samodzielnego wykonania. Złożoność algorytmu w tym przypadku jest trudno określić, gdyż nie wiemy, jak ta funkcja jest zrealizowana w arkuszu. W najgorszym przypadku ta złożoność będzie ograniczona przez złożoność porządkowania.

Zadanie Kodowanie liczb

(Próbny egzamin maturalny z informatyki, Arkusz I, OKE Warszawa, październik 2004)

a) Jaką największą dodatnią liczbę dwójkową można przedstawić za pomocą N cyfr (N — dowolna liczba naturalna)?

Wpisz odpowiedź

b) Dane są dwie liczby binarne $A = (1001\ 1000)_2$ i $B = (1001)_2$

Oblicz $A + B$, $A - B$, $A * B$

Wynik podaj w kodach dwójkowym i szesnastkowym.

Działanie BIN HEX

$$A + B$$

$$A - B$$

$$A * B$$

KOMENTARZ. W punkcie a) należy zauważyć, że największa N -cyfrowa liczba dwójkowa ma same jedynki, a jedynki odpowiadają potęgom liczby 2. Należy je więc zsumować (porównaj końcowy fragment punktu 7.1). W punkcie b) należy posłużyć się arytmetyką binarną, a przy zamianie liczb binarnych na szesnastkowe warto skorzystać z zadania 7.3 i problemu 7.1.

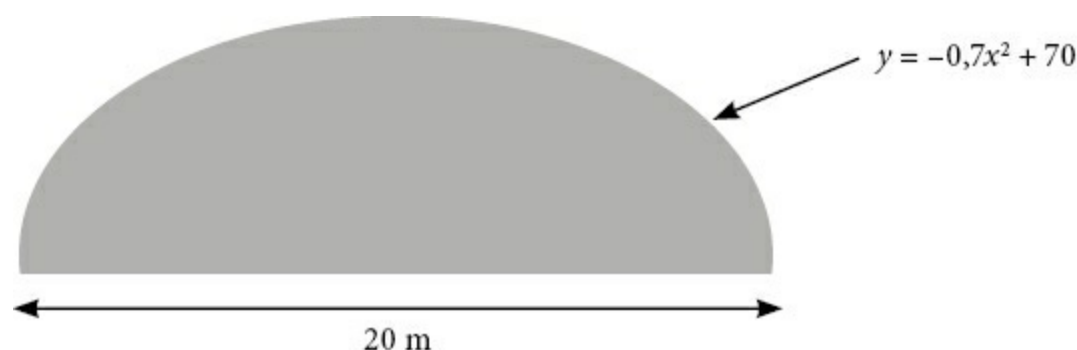
Zadanie Kopiec

(Próbny egzamin maturalny z informatyki, Arkusz I, UMK Toruń, grudzień 2005)

Postanowiono usypać kopiec, którego każdy pionowy przekrój poprzeczny opisuje równanie

$$y = -0,7x^2 + 70.$$

Podstawa kopca ma średnicę 20 m. Oblicz, z dokładnością d podaną w m^3 , ile ziemi potrzeba na usypanie kopca. Liczba rzeczywista d ma być czytana z klawiatury.



Wykonaj następujące zadania:

- narysuj schemat blokowy algorytmu do wyznaczania objętości kopca,
- podaj kryterium użyte do określenia dokładności wyznaczenia objętości kopca,
- scharakteryzuj zastosowany do rozwiązania algorytm,
- napisz program wyznaczający, z podaną dokładnością, ile ziemi potrzeba na usypanie kopca. Program czyta dane (liczbę d) z klawiatury i wynik zapisuje do pliku *kopiec.txt*,

e) podaj wynik działania programu otrzymany dla dokładności: 5 m^3 , 2 m^3 :

Dokładność Wynik [m3]

5 m^3

2 m^3

Do oceny oddajesz plik *kopiec.txt*, zapisany na nośniku WYNIKI, zawierający kompletny program (z funkcjami i procedurami), napisany w wybranym przez Ciebie języku.

Pamiętaj, że ocenie podlega również styl programowania (odpowiednie nazewnictwo zmiennych, stosowanie niezbędnych komentarzy i wcięć w zapisie kodu).

KOMENTARZ. Zadanie to jest rozwinięciem zadania, w którym obliczamy powierzchnię obszaru ograniczonego krzywą o podanym wzorze, porównaj punkt 2.7 w [Inf-II], oraz zamieszczone tam rozwiązanie innego zadania maturalnego.

Zadanie Co robi ten algorytm?

(Egzamin maturalny z informatyki, Arkusz I, sesja zimowa, CKE, styczeń 2006)

Przeanalizuj działanie poniższego algorytmu, jeżeli tablica A zawiera n liczb całkowitych z zakresu $[0, k]$.

```
1 for  $i \leftarrow 0$  to  $k$  do  $B[i] \leftarrow 0$ ;  
2  $pozycja \leftarrow 0$ ;  
3 for  $i \leftarrow 1$  to  $n$  do  $B[A[i]] \leftarrow B[A[i]] + 1$ ;  
4 for  $i \leftarrow 0$  to  $k$  do  
5   for  $j \leftarrow 1$  to  $B[i]$  do  
6   begin  $pozycja \leftarrow pozycja + 1$ ;  $A[pozycja] \leftarrow i$  end;
```

a) Uzupełnij tabelę — określ typy zmiennych: i , j , A , B , $pozycja$ i opisz ich przeznaczenie:

Zmienna Typ Przeznaczenie

i, j

A

B

$Pozycja$

b) Opisz znaczenie czynności wykonywanych w wierszach o numerach: 3,4 – 6.

c) Uzupełnij podane niżej zdania:

Tablica B jest tablicą pomocniczą. Jeśli tablica A zawiera n liczb z zakresu $[0, k]$, to tablica B zawiera liczb z zakresu

Dla $A = [1, 2, 4, 2, 0]$, po wykonaniu algorytmu, tablica $B = [. , , , ,]$.

Z uwagi na konieczność zastosowania dodatkowej tablicy, powyższego algorytmu nie można określić mianem

d) Przeprowadź analizę złożoności czasowej algorytmu i uzupełnij poniższy wniosek.

Założmy, że k jest ustalone, np. zawsze równe 5. Wówczas:

► złożoność czasowa przedstawionego algorytmu ma charakter (podkreśl prawidłową odpowiedź):

liniowy, kwadratowy, sześcienny, wykładniczy;

► symbolicznie złożoność taką można zapisać jako

KOMENTARZ. Algorytm, o którym jest mowa w tym zadaniu, nosi nazwę porządkowania przez zliczanie (porównaj problem 13.25). Jest to szczególna wersja algorytmu kubełkowego (porównaj punkt 6.3.1). Złożoność tych algorytmów jest proporcjonalna do liczby elementów w porządkowanym ciągu.

Zadanie Suma silni

(Egzamin maturalny z informatyki, Arkusz I, 2006)

Pojęcie silni dla liczb naturalnych większych od zera definiuje się następująco:

$n! = 1$, dla $n = 1$,

$n! = (n - 1)! * n$, dla $n > 1$.

Rozpatrzmy funkcję $ss(n)$ zdefiniowaną następująco:

$ss(n) = 1! + 2! + 3! + 4! + .. + n!, (*)$

gdzie n jest liczbą naturalną większą od zera.

a) Podaj, ile mnożeń trzeba wykonać, aby obliczyć wartość funkcji $ss(n)$, korzystając wprost z podanych wzorów, tzn. obliczając każdą silnię we wzorze (*) oddzielnie.

Uzupełnij poniższą tabelę.

Wartość funkcji Liczba mnożeń

ss(3)

ss(4)

ss(n)

b) Zauważmy, że we wzorze na $ss(n)$, czynnik 2 występuje w $n - 1$ silniach, czynnik 3 w $n - 2$ silniach,..., czynnik n w 1 silni. Korzystając z tej obserwacji, przekształć wzór funkcji $ss(n)$ tak, aby można było policzyć wartość $ss(n)$, wykonując dokładnie $n - 2$ mnożenia dla każdego $n \geq 2$. Uzupełnij poniższą tabelę.

Wartość funkcji		Przekształcony wzór	Liczba mnożeń
ss(1)	1		0
ss(2)	1 + 2		0
ss(3)	1 + 2 * (1 + 3)		1
ss(4)	1 + 2 * (1 + 3 * (1 + 4))		2
ss(5)			
ss(n)		$1 + 2 * (1 + 3 * (1 + \dots (n - 2) * (\text{ }) \dots))$	$N - 2$

Zapisz, w wybranej przez siebie notacji (lista kroków, schemat blokowy lub język programowania), algorytm obliczania wartości funkcji $ss(n)$ zgodnie ze wzorem zapisanym przez Ciebie w tabeli. Podaj specyfikację dla tego algorytmu, czyli dane, wynik i algorytm.

KOMENTARZ. Sposób obliczania wartości $ss(n)$ przypomina schemat Hornera.

Zadanie Liczby pierwsze

(Egzamin maturalny z informatyki, Arkusz I, 2006)

Poniżej przedstawiono algorytm wyznaczający wszystkie liczby pierwsze z przedziału $[2, N]$, wykorzystujący metodę Sita Eratostenesa. Po zakończeniu wykonywania tego algorytmu, dla każdego $i = 2, 3, \dots, N$, zachodzi $T[i] = 0$, gdy i jest liczbą pierwszą, natomiast $T[i] = 1$, gdy i jest liczbą złożoną.

Dane: Liczba naturalna $N \geq 2$.

Wynik: Tablica $T[2.. N]$, w której $T[i] = 0$, gdy i jest liczbą pierwszą, natomiast $T[i] = 1$, gdy i jest liczbą złożoną.

Krok 1. Dla $i = 2, 3, \dots, N$ wykonuj $T[i] := 0$.

Krok 2. $i := 2$.

Krok 3. Jeżeli $T[i] = 0$, to przejdź do kroku 4., w przeciwnym razie przejdź do kroku 6.

Krok 4. $j := 2 * i$.

Krok 5. Dopóki $j \leq N$, wykonuj

$T[j] := 1$,

$j := j + i$.

Krok 6. $i := i + 1$.

Krok 7. Jeżeli $i < N$, to przejdź do kroku 3., w przeciwnym razie zakończ wykonywanie algorytmu.

Uwaga: „ $:=$ ” oznacza instrukcję przypisania.

a) Dane: liczba naturalna $M \geq 1$ i tablica $A[1..M]$ zawierająca M liczb naturalnych z przedziału $[2, N]$. Korzystając z powyższego algorytmu, zaprojektuj algorytm, wyznaczający te liczby z przedziału $[2, N]$, które nie są podzielne przez żadną z liczb $A[1], \dots, A[M]$. Zapisz go w wybranej przez siebie notacji (lista kroków, schemat blokowy lub język programowania) wraz ze specyfikacją.

b) Do algorytmu opisanego na początku zadania wprowadzamy modyfikacje, po których ma on następującą postać:

Krok 1. Dla $i = 2, 3, \dots, N$ wykonuj $T[i] := 0$.

Krok 2. $i := 2$.

Krok 3. Jeżeli $T[i] = 0$, to przejdź do kroku 4., w przeciwnym razie przejdź do kroku 6.

Krok 4. $j := 2 * i$.

Krok 5. Dopóki $j \leq N$, wykonuj

$T[j] := T[j] + 1$,

$j := j + i$.

Krok 6. $i := i + 1$.

Krok 7. Jeżeli $i < N$, to przejdź do kroku 3., w przeciwnym razie zakończ wykonywanie algorytmu.

Podaj, jakie będą wartości $T[13]$, $T[24]$, $T[33]$ po uruchomieniu tak zmodyfikowanego algorytmu dla $N = 100$.

Podaj, dla jakiej wartości $T[i]$, dla i z przedziału $[2, N]$, i jest liczbą pierwszą.

Napisz, jaką własność liczb $i = 2, \dots, N$ określają wartości $T[i]$ po wykonaniu tak zmodyfikowanego algorytmu.

c) Sito Eratostenesa służy do wyznaczania wszystkich liczb pierwszych z danego przedziału $[2, N]$. Podaj, w wybranej przez siebie notacji (lista kroków, schemat blokowy lub język programowania), inny algorytm, który sprawdza, czy podana liczba naturalna $L > 1$ jest liczbą pierwszą. Zauważ, że chcemy sprawdzać pierwszość tylko liczby L , natomiast nie jest konieczne sprawdzanie pierwszości liczb mniejszych od L . Przy ocenie Twojego algorytmu będzie brana pod uwagę jego złożoność czasowa.

Specyfikacja

Dane: Liczba naturalna $L > 1$.

Wynik: Komunikat „Tak”, jeśli L jest liczbą pierwszą, komunikat „Nie” w przeciwnym razie.

KOMENTARZ. Zauważ, że algorytm zmodyfikowany w punkcie b) różni się od algorytmu z punktu a) tylko jedną instrukcją w kroku 5. Zastanów się, co oznacza ta zmiana, czyli jaką interpretację ma wartość $T[i]$ — jest to pytanie z końca punktu b). Odnośnie punktu c) — o sicie Eratostenesa piszemy w punkcie 7.6.2, a rozwiązanie dla tego punktu jest modyfikacją algorytmu podanego w punkcie 7.6.1. Patrz również rozdział 8. w książce [Piramidy].

Zadanie Wypłata

(Próbny egzamin maturalny z informatyki, Arkusz I, CKE, grudzień 2006)

Pracownicy pewnego zakładu pracy otrzymują pensje w kwotach będących wielokrotnością 10 złotych. Kasjer, przygotowując wypłatę, przed pobraniem pieniędzy z banku musi obliczyć, ile potrzebuje banknotów o poszczególnych nominałach (10 zł, 20 zł, 50 zł, 100 zł, 200 zł) do zrealizowania wypłaty. Kasjer każdemu pracownikowi chce wypłacić pensję w możliwie najmniejszej liczbie banknotów.

Przyjmijmy, że kwoty wypłat dla poszczególnych pracowników są podane w n -elementowej tablicy WYPŁATY[1 ... n], gdzie n jest liczbą pracowników zakładu. Zaproponuj algorytm obliczania liczby banknotów w poszczególnych nominałach, które kasjer musi pobrać z banku. Wynik obliczeń należy umieścić w tablicy LICZBY[1..5], gdzie:

LICZBY[1] to liczba banknotów o nominale 200 zł,

LICZBY[2] to liczba banknotów o nominale 100 zł,

LICZBY[3] to liczba banknotów o nominale 50 zł,

LICZBY[4] to liczba banknotów o nominale 20 zł,

LICZBY[5] to liczba banknotów o nominale 10 zł.

Podaj specyfikację algorytmu i zapisz go w wybranej przez siebie notacji (lista kroków, schemat blokowy, język programowania).

KOMENTARZ. Rozwiązanie tego zadania otrzymujemy jako iterację rozwiązania problemu reszty dla kwot zapisanych w tablicy WYPŁATY, porównaj zadanie 11.4.

Zadanie Dziwny ciąg

(Próbny egzamin maturalny z informatyki, Arkusz I, CKE, grudzień 2006)

Rozważamy ciąg liczb naturalnych $D(n)$ dla $n = 0, 1, 2, \dots$, zdefiniowany następująco:

$$D(n) = 1 \text{ dla } n = 0 \text{ lub } n = 1,$$

$$D(n) = D(n \operatorname{div} 4) + 1 \text{ dla parzystego } n > 1,$$

$$D(n) = D(3n + 1) + 1 \text{ dla nieparzystego } n > 1.$$

Uwaga: operator div oznacza dzielenie całkowite, np.: $3 \operatorname{div} 4 = 0$, $15 \operatorname{div} 2 = 7$, $9 \operatorname{div} 3 = 3$.

Na przykład:

$$D(5) = D(16) + 1 = D(4) + 2 = D(1) + 3 = 4.$$

a) Korzystając z powyższej definicji, oblicz $D(3)$, $D(17)$, $D(31)$. Zapisz poniżej swoje obliczenia.

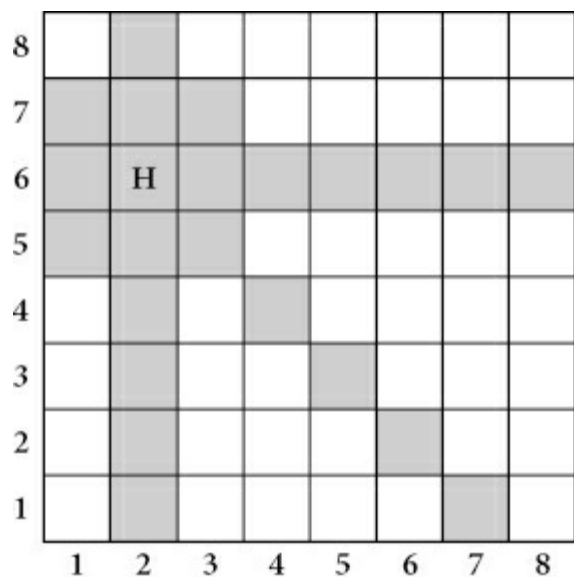
b) Przedstaw, w wybranej przez siebie notacji (lista kroków, schemat blokowy lub język programowania), **nierekurencyjny** algorytm obliczania wartości $D(n)$ dla danej liczby naturalnej n . Podaj specyfikację tego algorytmu.

KOMENTARZ. Ciąg $D(n)$ jest podobnie definiowany, jak ciągi w punkcie 12.3. Algorytm, o który chodzi w punkcie b), polega na obliczaniu kolejnych wartości $D(k)$ dla $k = 1, 2, 3, \dots, n$ poprzez odwoływanie się do poprzednich wartości tego ciągu. Jest to możliwe, gdyż jeśli n jest liczbą parzystą, to indeks elementu ciągu po prawej stronie wzoru na $D(n)$ maleje co najmniej 4 razy, a jeśli n jest liczbą nieparzystą, to w dwóch kolejnych krokach indeks ten również maleje.

Zadanie Szachownice

(Egzamin maturalny z informatyki, Arkusz I, CKE, maj 2007)

Zgodnie z regułami gry w szachy, hetman (królowa) może atakować figury ustawione na polach w kolumnie, wierszu oraz dwóch przekątnych przechodzących przez pole, w którym jest ustawiony. O tych polach mówimy, że są atakowane przez hetmana.



Na rysunku hetman stoi w polu (2, 6) i atakuje $(7 + 7 + 6 + 3) = 23$ pola. Zostały one zamalowane kolorem szarym.

a) Poniżej znajduje się tabela o wymiarach 5×5 . Korzystając z powyższej obserwacji, uzupełnij pola tabeli, wpisując do każdego z nich liczbę pól, które atakowałby hetman znajdujący się w tym polu. Hetman stojący w polu (1, 1) atakuje 12 pól planszy.

5					
4					
3					
2					
1	12				
	1	2	3	4	5

b) Określ liczbę atakowanych pól na szachownicy 32×32 , gdy dane są współrzędne ustawienia hetmana.

Dla (2, 2) wynik =

Dla (5, 4) wynik =

Dla (20, 18) wynik =

Dla (25, 30) wynik =

c) Podaj specyfikację i zapisz algorytm (w postaci listy kroków, schematu blokowego lub w języku programowania), który dla dowolnej dodatniej liczby

całkowitej $n \leq 50$ i położenia hetmana (x, y) na szachownicy o wymiarach $n \times n$, gdzie $1 \leq x, y \leq n$, pozwoli obliczyć liczbę pól atakowanych przez tego hetmana.

KOMENTARZ. Aby podać algorytm w punkcie c), należy najpierw określić warunek dla pola szachownicy (i, j) , aby było ono szachowane przez hetmana stojącego na pozycji (x, y) . Należy również uwzględnić położenie pola (x, y) względem brzegów szachownicy.

Zadanie Sumy

(Egzamin maturalny z informatyki, Arkusz I, CKE, maj 2007)

W tabeli podany jest algorytm, który pozwala obliczyć wartość pewnej *sumy* dla danej dodatniej liczby całkowitej n .

1 $p1 \leftarrow 1$

2 $suma \leftarrow 0$

3 dla $k \leftarrow 1..n$ wykonuj

4 $p1 \leftarrow p1 * n$

5 $p2 \leftarrow 1$

6 dla $i \leftarrow 1..n$ wykonuj

7 $p2 \leftarrow p2 * k$

8 $suma \leftarrow suma + p1 + p2$

3.1. Podaj, jaką wartość przyjmie zmienna $p1$ w wyniku działania powyższego algorytmu dla $n = 3$. $p1 =$

3.2. Podaj, jaką wartość przyjmie zmienna $p2$ w wyniku działania powyższego algorytmu dla $n = 3$. $p2 =$

3.3. Podaj, jaką wartość przyjmie zmienna $suma$ w wyniku działania powyższego algorytmu dla $n = 3$. $suma =$

3.4. Zakreślając właściwą odpowiedź, zaznacz, jaką wartość przyjmie zmienna $suma$ w wyniku działania powyższego algorytmu.

a) $\sum_{k=1}^n (k^k + n^2)$ b) $\sum_{k=1}^n (n^n + k^n)$ c) $\sum_{k=1}^n (n^k + k^2)$

d) $\sum_{k=1}^n (n^k + k^n)$ e) $\sum_{k=1}^n (n^n + k^k)$

gdzie $\sum_{k=1}^n a_n = a_1 + a_2 + \dots + a_n$

3.5. Zakreślając właściwą odpowiedź, podaj, ile wynosi liczba operacji arytmetycznych (dodawania i mnożeń) wykonywanych w czasie realizacji przedstawionego algorytmu.

a) $3n$ b) $n^2 + 3n$ c) $2n + n^2$

d) $nn + 2n$ e) $n! + 2n$

3.6. Zmień wiersze 6. i 7. w rozważanym algorytmie w taki sposób, aby po jego wykonaniu wartością zmiennej *suma* było $\sum_{k=1}^n (n^k + k!)$.

KOMENTARZ. W tym zadaniu należy dobrze zinterpretować działanie instrukcji iteracyjnej.

Zadanie Liczby superpierwsze

(Egzamin maturalny z informatyki, Arkusz I, CKE, maj 2007)

Liczba **superpierwsza** to taka liczba naturalna, która spełnia następujące warunki:

► jest liczbą pierwszą;

► suma cyfr tej liczby jest również liczbą pierwszą.

Liczba **super B pierwsza**, oprócz wymienionych dwóch warunków, spełnia warunek trzeci:

► suma cyfr w jej zapisie binarnym jest także liczbą pierwszą.

a) Dla każdego z podanych niżej przedziałów oblicz, ile jest liczb super B pierwszych w tym przedziale. Wyniki wpisz do tabeli. Dodatkowo, w plikach o nazwach *1.txt*, *2.txt* i *3.txt*, zapisz wszystkie liczby super B pierwsze odpowiednio z przedziałów 1., 2. i 3., po jednej liczbie w każdym wierszu.

Nr przedziału	Przedział	Liczba wystąpień liczb super B pierwszych w przedziale
---------------	-----------	--

1. [2, 1000]
2. [100, 10000]
3. [1000,
100000]

b) Odpowiedz na następujące pytania:

Ile jest liczb w przedziale [100, 10000], których suma cyfr jest liczbą pierwszą?

Czy suma wszystkich liczb super B pierwszych z przedziału [100, 10000] jest liczbą pierwszą?

Do oceny oddajesz plik(i) zawierający(e) komputerową(e) realizację(e) rozwiązania zadania oraz pliki *1.txt*, *2.txt* i *3.txt*.

KOMENTARZ. Przy rozwiązaniu tego zadania przydatny jest algorytm badania, czy dana liczba jest liczbą pierwszą, porównaj punkt 7.6.1 i punkt 8.1 w książce [Piramidy]. Ponadto do rozkładu liczby na postać binarną można posłużyć się algorytmem z punktu 7.1.

[1] W treści niektórych zadań poczyniono skróty, bez szkody dla zagadnień algorytmicznych, których dotyczą. Pełne sformułowanie zadań maturalnych można znaleźć w serwisie CKE na stronie www.cke.edu.pl.

Spis treści

Od autora...

...do uczniów

...do nauczyciela

...podziękowania

Wstęp do wydania Helion

Wyróżnienia i oznaczenia w tekście

Rozdział 1. Algorytmy i sposoby ich przedstawiania

1.1. Algorytm w procesie powstawania

1.2. Algorytmy na przestrzeni wieków

1.3. Reprezentacje problemów i algorytmów

1.4. Ćwiczenia, zadania, problemy

Rozdział 2. Algorytmy liniowe

2.1. Zadania

Rozdział 3. Algorytmy z rozgałęzieniami

3.1. Rozwiązywanie równania kwadratowego

3.2. Rozwiązywanie równania liniowego

3.3. Rozwiązywanie układu równań liniowych

3.4. Zadania

Rozdział 4. Porządkowanie kilku liczb

4.1. Porządkowanie trzech liczb

4.2. Porządkowanie czterech liczb

4.3. Porządkowanie pięciu liczb

4.4. Zadania i problemy

Rozdział 5. O czym mówią dane — algorytmy iteracyjne

5.1. Reprezentowanie i przeszukiwanie zbioru

5.2. Obliczanie średniej

5.3. Znajdowanie największego elementu

5.4. Kompletowanie podium zwycięzców

5.5. Znajdowanie jednocześnie największego i najmniejszego elementu

5.6. Obliczanie innych miar centralności danych

5.7. Zadania i problemy

Rozdział 6. Porządkowanie ciągu elementów

6.1. Algorytm bąbelkowy

6.2. Porządkowanie przez wybór

6.3. Porządkowanie kubełkowe i pozycyjne

6.4. Zadania i problemy

Rozdział 7. Inne algorytmy iteracyjne — schemat Hornera, algorytm Euklidesa, sito Eratostenesa

7.1. Zapisywanie liczb w systemie binarnym

[7.2. Schemat Hornera](#)

[7.3. Zastosowania schematu Hornera](#)

[7.4. Algorytm Euklidesa](#)

[7.5. Zastosowania algorytmu Euklidesa](#)

[7.6. Liczby pierwsze i liczby złożone](#)

[7.7. Obliczanie wartości pierwiastka kwadratowego](#)

[7.8. Zadania i problemy](#)

[Rozdział 8. Algorytmy rekurencyjne](#)

[8.1. Inne spojrzenie na iterację](#)

[8.2. Problemy z rekurencyjną naturą](#)

[8.3. Zadania i problemy](#)

[Rozdział 9. Dziel i zwyciężaj](#)

[9.1. Rekurencyjne znajdowanie największego i najmniejszego elementu](#)

[9.2. Przeszukiwanie binarne, czyli przez połowienie](#)

[9.3. Przeszukiwanie interpolacyjne](#)

[9.4. Znajdowanie miejsca zerowego funkcji metodą połowienia przedziału](#)

[9.5. Zadania i problemy](#)

[Rozdział 10. Porządkowanie ciągu elementów](#)

[10.1. Porządkowanie przez umieszczanie](#)

[10.2. Porządkowanie przez scalanie](#)

[10.3. Szybki algorytm porządkowania](#)

[10.4. Własności algorytmów porządkowania](#)

[10.5. Zadania i problemy](#)

[Rozdział 11. Wychodzenie z labiryntu i pakowanie plecaka](#)

[11.1. Znajdowanie wyjścia z labiryntu](#)

[11.2. Pakowanie najcenniejszego plecaka](#)

[11.3. Zadania i problemy](#)

[Rozdział 12. Własności algorytmów — podsumowanie](#)

[12.1. Algorytmy — spojrzenie z lotu ptaka](#)

[12.2. Poprawność algorytmów](#)

[12.3. Skończoność algorytmów](#)

[12.4. Złożoność i efektywność algorytmów](#)

[12.5. Zadania i problemy](#)

[Rozdział 13. Problemy](#)

[13.1. Problemy łatwiejsze](#)

[13.2. Problemy trudniejsze](#)

[Rozdział 14. Gdzie szukać dalszych informacji o algorytmach](#)

[14.1. Opracowania podstawowe](#)

[14.2. Opracowania zaawansowane](#)

[Rozdział 15. Algorytmika w zadaniach maturalnych](#)