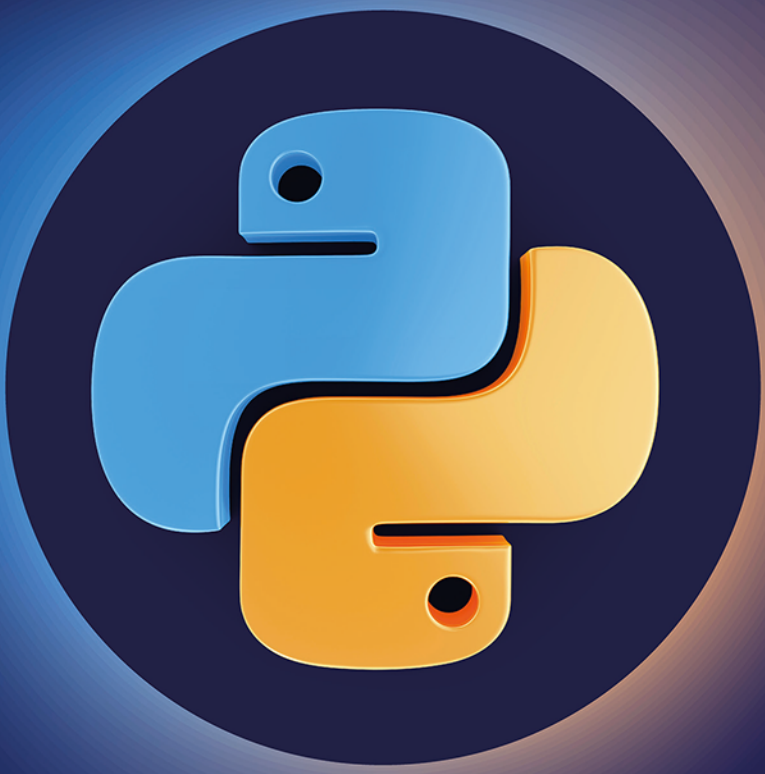
Piotr Wróblewski

Algorytmy w Pythonie

TECHNIKI PROGRAMOWANIA DLA PRAKTYKÓW



Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/algpyt
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
https://ftp.helion.pl/przyklady/algpyt.zip

ISBN: 978-83-283-9368-4

Printed in Poland.

Spis treści

Lepiej
Lepiej bez przerwy
Iść po grudzie?
Czy
Ciągle pod górę?

Himalaje
Każdy
Ma
Swoje
Himalaje

Dariusz Brzóska-Brzóskiewicz
z tomu *499 Haiku*
(wyd. Państwowy Instytut Wydawniczy, Warszawa 2020)

Przedmowa

Książka, którą trzymasz w ręku, jest przeznaczona zarówno dla uczniów i studentów potrzebujących prostego i praktycznego podręcznika podstaw algorytmiki opartego na Pythonie, jak i dla szerokiego grona zwolenników samego języka Python, którzy chcieliby wejść na nieco inny poziom programowania, bardziej „świadomego", a nie opartego na klasycznym zmaganiu się z komputerem i tworzeniu kodu metodą prób i błędów.

Jest to nowy tytuł, ale jego koncepcja posiada pewną historię. W ofercie wydawnictwa Helion jest od wielu lat dostępny tytuł *Algorytmy, struktury danych i techniki programowania*, który okazał się dużym sukcesem wydawniczym, o czym świadczy aż sześć wydań w Polsce i kilka zagranicznych. Językiem programowania użytym do ilustrowania algorytmów nie był tzw. pseudokod, tylko prawdziwy, dający się kompilować i uruchamiać kod w C++. W 2019 r., w reakcji na dużą popularność języka Java, w wydawnictwie Helion zapadła decyzja o wydaniu pozycji *Algorytmy, struktury danych i techniki programowania dla programistów Java*, której zaletą było uwzględnienie różnic wynikających z cech tego języka, mających duży wpływ na sposoby modelowania danych i realizacji algorytmów.

Kierując się potrzebami czytelników, postanowiliśmy teraz dostarczyć podobny prezent miłośnikom Pythona. Ten język nie bez powodu stał się niesłychanie popularny w środowisku zawodowych i amatorskich programistów, niedawno wszedł także do podstawy programowej nauczania informatyki w liceum! Jego cechy to:

- prosta i czytelna składnia, bez uciążliwej i znanej z innych języków „ornamentowej" otoczki klas wywoławczych, kłopotliwej czasami funkcji `main` itp.;
- niska bariera wejścia (szybciej się nauczysz Pythona niż niuansów C++ lub Javy);
- łatwa instalacja (wszelkie narzędzia potrzebne do efektywnej pracy instalujesz z Internetu w kilka minut);
- możliwość testowania „na żywo", bez konieczności kompilowania kodu do postaci binarnej (jest to język skryptowy, pracujący w trybie interpretera).

Python króluje również w rankingach branżowych, np. w ciągu ostatnich dwóch lat firma analityczna RedMonk po raz kolejny potwierdziła dominującą pozycję tego języka. W jej raporcie *https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/* umieszczono go na drugim miejscu, wyżej niż Javę!

Prostota języka ma jednak swoje konsekwencje i używanie Pythona może się wiązać z pewną pułapką: nowy adept tego języka, jakże często taki, który nie posiada wykształcenia informatycznego, nierzadko i na pewno zbyt szybko przechodzi od pomysłu do realizacji, wpadając w cykl, który na początku książki nieco ironicznie zilustrowałem cytatami z haiku Dariusza Brzóski-Brzóskiewicza — ciężka i mozolna praca, wyzwania nie do pokonania, frustracja zamiast radości tworzenia…

Warto zatem pamiętać, że Python, tak jak każdy język sterujący komputerem, to nie tylko pewna składnia i gotowe biblioteki użytkowe! Jeśli zechcesz go używać nieco bardziej świadomie i bez zbędnego stresu, warto zainwestować nieco w teorię informatyki i zanurzyć się w przebogatym świecie typów i struktur danych oraz algorytmów.

Jest tylko jeden problem: Python jest idealny dla praktyków, a dla nich każda teoria jest zwyczajnie nudna! Dlatego, podobnie jak w moich innych książkach, skoncentruję się na praktyce programowania, będziemy omawiali prawdziwy kod dający się uruchomić, a wszelki bagaż teoretyczny zostanie przemycony niejako między wierszami i ograniczony do minimum.

Niezależnie jednak od celu, jaki Ci przyświeca, sam język musisz najpierw skutecznie poznać, w czym postaram się… także skutecznie (mam nadzieję) Ci pomóc!

Pisząc tę książkę, zastosowałem podejście, które sprawdziło się w kilku innych moich poprzednio wydanych książkach:

- Postaram się przede wszystkich nie zanudzić.
- Znajdziesz tutaj minimum teorii i maksimum praktyki.
- Przetestujemy przykładowy, gotowy do użycia kod, który może posłużyć jako punkt wyjścia do rozwijania większych aplikacji.

Mam zatem nadzieję, że ten tytuł, podobnie jak inne moje książki, także wzbudzi zainteresowanie czytelników!

Przykładowe programy w Pythonie

Zdając sobie sprawę, że Python może być pewnej grupie czytelników nieznany, na kartach książki „przemyciłem" miniwykład z obszaru jego podstawowych struktur składniowych. Nie chciałem jednak zamieniać książki w podręcznik Pythona, ale dość często w trakcie lektury — w tekście książki lub w formie komentarzy do prezentowanych w niej listingów — znajdziesz omówienie wybranych elementów tego języka, tak aby móc je przeanalizować i zrozumieć, jak skutecznie używać Pythona do rozwiązywania zagadnień algorytmicznych.

Ponadto, korzystając z dodatku A, nauczysz się skutecznie pracować z samym środowiskiem Pythona, zaczynając od poprawnej instalacji środowiska uruchomieniowego.

Wszystkie omawiane w książce programy zostały umieszczone na serwerze FTP wydawnictwa Helion, pod adresem *https://ftp.helion.pl/przyklady/algpyt.zip*. Zawarte tam pliki źródłowe są zazwyczaj pełniejsze i bardziej rozbudowane niż warianty prezentowane w wersji drukowanej. Można zatem założyć, że jeśli w trakcie wykładu prezentowana jest jakaś funkcja bez podania *explicite* sposobu jej użycia, to na pewno wersja źródłowa zawiera reprezentacyjny przykład jej

zastosowania. Warto zatem podczas lektury porównywać wersje umieszczone na FTP z tymi, które zostały omówione na kartach książki!

W ramach dodatkowego ułatwienia, w celu zidentyfikowania, gdzie w książce jest opisywany plik źródłowy dostępny na serwerze FTP, możesz posłużyć się skorowidzem.

Mój podręcznik nie zastąpi oczywiście książki poświęconej samemu Pythonowi, będzie jednak przydatny osobom, które nie stawiają sobie za cel dogłębnego poznawania nowego języka. Jeśli odczujesz potrzebę głębszego poznania samego języka, sięgnij po inne pozycje z oferty Helionu lub nawet po moją książkę *Python dla testera* (*https://helion.pl/ksiazki/pyttes.htm*). Zawiera ona praktyczny podręcznik języka, który jeden z czytelników-recenzentów określił zabawnym sformułowaniem „od zera do bohatera". Tak przy okazji, to tytuł tej pozycji jest nieco mylący, z zawartych tam treści powinni skorzystać wszyscy, którzy chcą poznać Pythona od nieco bardziej praktycznej strony, nie tylko testerzy!

Co odróżnia tę książkę od innych podręczników?

Klasyczne podręczniki algorytmiki, zwłaszcza te najbardziej renomowane i napisane przez „ojców-założycieli" współczesnej informatyki, są wartościowe, ale często operują mniej popularnymi językami (np. Pascalem) albo są przesycone aparatem matematycznym i używają pseudokodu, a nie prawdziwego języka programowania. Na pewno nie ułatwia to lektury czytelnikom pragnącym nie tylko zrozumieć opisywane zagadnienia, ale chcącym szybko zastosować je w praktyce. W związku z tym założyłem prosty język publikacji: unikałem w miarę możliwości zagłębiania się w złożony aparat matematyczny, ilustrując jednocześnie materiał teoretyczny przykładami i prawdziwym kodem dającym się łatwo uruchomić i który można samodzielnie modyfikować. Jest to ukłon w stronę czytelników znużonych publikacjami nasyconymi pseudokodem, niedającym się łatwo przełożyć na realia kompilatorów i wymogów systemów operacyjnych. Kody źródłowe programów w tej książce są zawsze gotowe do uruchomienia i zostały przetestowane w najnowszej edycji Pythona w kilku systemach operacyjnych (głównie w Windowsie i macOS-ie, częściowo w Linuksie).

Szereg praktycznych porad dotyczących procesu instalacji i używania środowiska Pythona pozwoli płynnie przejść od teorii do praktyki. Większość teorii jest poparta łatwymi do analizy przykładowymi programami, które można skompilować i sprawdzić w kilka minut. Przy pewnej dyscyplinie wewnętrznej, wyposażony w taki oręż, bez problemu poradzisz sobie z kolejno omawianymi zagadnieniami.

Mam oczywiście świadomość, że w obrębie jednej książki nie jest możliwe zaprezentowanie wszystkiego, co najważniejsze w dziedzinie algorytmiki. Jest to niewykonalne z uwagi na rozległość dziedziny, z jaką mamy do czynienia. Może się więc

okazać, że to, co zostało pomyślane jako logicznie skonstruowana całość, jednych rozczaruje, innych zaś przytłoczy ogromem poruszanych zagadnień. Moim pragnieniem było stworzenie w miarę reprezentatywnego przeglądu zagadnień algorytmicznych, przydatnego dla tych czytelników, którzy programowanie w Pythonie mają zamiar potraktować w sposób profesjonalny. Po przeczytaniu tej książki być może odczujesz, Drogi Czytelniku, potrzebę przejrzenia innej podobnej literatury (spis znajdziesz na końcu książki), ale nie zdziw się, jeśli spotkasz pozycje przeładowane matematyką lub zawierające masę pseudokodu zamiast prawdziwych programów, które dają się łatwo kompilować i testować. Mam jednak nadzieję, że po lekturze tego podręcznika będzie Ci łatwiej zmagać się z bardziej opasłymi tomami, niestety zazwyczaj pisanymi dość hermetycznym językiem (np. monumentalna publikacja [CLRS12] liczy ponad 1300 stron i waży około 2 kg…). Na szczęście w ostatnich latach pojawiły się już podręczniki algorytmiki ilustrowane kodem Pythona; jeśli zechcesz zanurzyć się w uniwersum algorytmów, używając tego języka, zajrzyj do [HML14] lub [GM13].

Mój podręcznik polecam szczególnie osobom zainteresowanym **praktycznym programowaniem**, a niemającym do tego solidnych podstaw teoretycznych. Ponieważ obejmuje on dość obszerną klasę zagadnień z dziedziny informatyki, będzie również użyteczny jako repetytorium dla tych, którzy zajmują się programowaniem zawodowo. Jest to jednak książka dla osób, które przynajmniej częściowo zetknęły się z programowaniem i rozumieją podstawowe pojęcia, takie jak zmienna, program, kompilacja, bo tego typu terminy stanowią podstawę języka używanego w tej książce. Nie będę się zagłębiał w ich wyjaśnianie, z wyjątkiem sytuacji dotyczących specyfiki realizacji pewnych mechanizmów w Pythonie.

Jak należy czytać tę książkę?

Jeśli już zetknąłeś się wcześniej z tematyką podejmowaną w tej książce, możesz czytać ją, Drogi Czytelniku, w dowolnej kolejności, wynikającej z bieżących potrzeb.

Początkującym zalecam trzymanie się porządku narzuconego przez układ rozdziałów; szczególnie ważne są te poświęcone strukturom danych i rekurencji. Zachęcam do aktywnej lektury połączonej z testowaniem przykładowych programów i rozwiązywaniem przykładowych zadań, bo nic tak nie utrwala wiedzy jak praktyczne spojrzenie na prezentowaną teorię (dla ułatwienia proponowane zadania są w dużej części rozwiązane, ewentualnie podane są szczegółowe wskazówki).

Do książki dodane zostały szczegółowy skorowidz i spis tabel oraz ilustracji — powinny one ułatwić odszukiwanie potrzebnych informacji merytorycznych oraz zidentyfikowanie tych miejsc w książce, w których jest opisywany plik źródłowy dostępny na FTP.

Co zostało opisane w tej książce?

Algorytmika stanowi gałąź wiedzy, która w ciągu ostatnich kilkudziesięciu lat dostarczyła wielu efektywnych narzędzi wspomagających rozwiązywanie różnorodnych zagadnień za pomocą komputera. Dla jednych jest to tylko swego rodzaju książka kucharska, do której sięga się, jeśli trzeba, po wybrane „przepisy". Dla innych algorytmika stanowi okazję do rozwinięcia umiejętności skutecznego rozwiązywania problemów i szkołę niestandardowego myślenia. Moją intencją w trakcie pisania książki było połączenie tych dwóch perspektyw poprzez prezentację w miarę szerokiego, ale zarazem pogłębionego zestawu tematów z tej dziedziny. Chciałem przy okazji opisywanych zagadnień ukazać odpowiednią perspektywę możliwych zastosowań komputerów, wychodząc poza wzory matematyczne i suchy kod przykładowych programów.

Aby ułatwić nieco nawigację po różnorodnych tematach poruszanych w książce, postanowiłem zaprezentować główną tematykę kolejnych rozdziałów z komentarzami dotyczącymi ich zawartości.

Rozdział 1. „Zanim wystartujemy"

Jest to rozbudowany wstęp pozwalający wziąć głęboki oddech przed przystąpieniem do pracy przy klawiaturze. W rozdziale tym poznasz kilka niezbędnych faktów historycznych dotyczących przeszłości algorytmiki i zrozumiesz, skąd wziął się obecny postęp w tej dziedzinie.

Rozdział 2. „Systemy obliczeniowe i podstawy kodowania"

Tu poznasz popularne systemy kodowania (np. dwójkowy, BCD, szesnastkowy) przydatne każdej osobie zainteresowanej programowaniem i modelowaniem informacji w algorytmach komputerowych. Celem tego rozdziału jest nie tylko omówienie teorii, ale i pokazanie, jak komputery kodują informacje zrozumiałe dla człowieka, choćby teksty i obrazy (np. format mapy bitowej BMP).

Rozdział 3. „Rekurencja"

Ten rozdział jest poświęcony jednemu z najważniejszych mechanizmów używanych w procesie programowania — rekurencji. Uświadamia zarówno oczywiste zalety, jak i nie zawsze widoczne wady tej techniki programowania. Zagadnienia poznasz zarówno na prostych, jak i trudnych przykładach oraz będziesz miał szansę się sprawdzić, rozwiązując ciekawe zadania graficzne.

Rozdział 4. „Analiza złożoności algorytmów"

Tu znajdziesz przegląd najpopularniejszych i najprostszych metod służących do wyznaczania sprawności obliczeniowej algorytmów i porównywania ich ze sobą w celu wybrania najefektywniejszego. Rozdział jest przeznaczony raczej dla studentów informatyki, choć osoby ogólnie zainteresowane programowaniem powinny nań rzucić choćby pobieżnie okiem, aby zrozumieć pojęcia używane w opisach

algorytmów i dzięki temu pojąć konsekwencje wyboru tej, a nie innej metody spośród katalogu dostępnych rozwiązań.

Rozdział 5. „Typy proste wbudowane”

Modelowanie danych zaczniemy od podstaw, czyli omówienia typów prostych i ich używania do modelowania nowych typów danych. Poznasz pojęcie referencji, zasady konwersji pomiędzy typami i niuanse dotyczące napisów w Pythonie.

Rozdział 6. „Modelowanie abstrakcyjnych struktur danych”

Poprzedni rozdział stanowi swego rodzaju rozgrzewkę przed właściwym tematem, jakim jest modelowanie abstrakcyjnych typów danych z użyciem szerokiego wachlarza technik realizacji różnorodnych struktur danych z uwzględnieniem możliwości oferowanych przez podejście obiektowe. Zobaczysz, że Python to dosłownie język z klasą!

Rozdział 7. „Przykładowe realizacje wybranych struktur danych”

W najbardziej programistycznie rozbudowanym rozdziale nauczysz się, jak można modelować złożone kolekcje danych w Pythonie. Poznane tu techniki programowania stanowią typowy wachlarz technik programistycznych. Zalecam nie tylko czytanie tego materiału, ale i aktywną pracę z kodem i dostosowywanie go do własnych potrzeb. Zanim sięgnie się po „gotowce”, które mogą być świetne w zawodowym tworzeniu oprogramowania, warto najpierw nauczyć się projektować własne struktury danych, a nabyte umiejętności z pewnością zaprocentują w przyszłości.

Rozdział 8. „Typy złożone wbudowane”

Dla „leniuchów”, którzy oczekują gotowych rozwiązań, przygotowałem uproszczony opis wybranych klas z biblioteki standardowej Pyhona, zawierającej szereg kompletnych realizacji skomplikowanych struktur danych, takich jak listy, zbiory i słowniki. Oczywiście oferta bibliotek Pythona (standardowych lub instalowanych osobno) jest zbyt obszerna, aby ująć ją rzetelnie jako jeden z tematów tej książki, dlatego postanowiłem ograniczyć się do wstępu praktycznego, ale na tyle reprezentatywnego, aby nawet początkujący programista mógł wspomnianej biblioteki użyć bez wnikania w pełne specyfikacje klas.

Rozdział 9. „Struktury danych o dostępie ograniczonym”

W tym rozdziale kontynuujemy omawianie typów danych charakteryzujących się brakiem swobodnego dostępu (np. stosów, kolejek, stert) — są one zatem przeciwieństwem np. tablic, gdzie każdy element jest dostępny przez adresowanie bezpośrednie. Po lekturze tego rozdziału przekonasz się, że w pewnych zastosowaniach takie konstrukcje mają sens i pozwalają znacznie łatwiej rozwiązywać wybrane problemy.

Rozdział 10. „Drzewa i ich reprezentacje”

W odrębnym rozdziale omawiam popularne realizacje struktur drzewiastych (np. drzewa binarne, binarne drzewa poszukiwań) i ich implementację programową. Szczególną uwagę poświęcam ukazaniu możliwych zastosowań nowo poznanych struktur danych, np. do modelowania wyrażeń arytmetycznych lub słowników.

Rozdział 11. „Algorytmy przeszukiwania”

W tym rozdziale wykorzystuję kilka poznanych już wcześniej metod w celu wyszukiwania elementów w słowniku, a następnie szczegółowo omawiam metodę transformacji kluczowej (ang. *hashing*) łącznie z pokazaniem własnej realizacji tablic hashujących.

Rozdział 12. „Algorytmy sortowania”

Tu poznasz najpopularniejsze i najbardziej znane procedury sortujące. Oczywiście w rozdziale nie wyczerpuję tematu — zakładam, że stanie się zachętą do dalszego studiowania arcyciekawej dziedziny algorytmów sortujących, mającej na dodatek duże walory dydaktyczne. Przedstawione są zarówno proste metody, np. sortowanie przez wstawianie oraz sortowanie bąbelkowe, jak i złożone, ze szczególnym naciskiem na dokładny opis metody Quicksort.

Rozdział 13. „Derekursywacja i optymalizacja algorytmów”

Tu prezentuję sposoby przekształcania programów rekurencyjnych na ich wersje iteracyjne. Rozdział ma charakter wybitnie techniczny i jest przeznaczony dla programistów zainteresowanych problematyką optymalizacji programów.

Rozdział 14. „Przeszukiwanie tekstów”

Ze względu na wagę i znaczenie tematu algorytmy przeszukiwania tekstów zostały zgrupowane w osobnym rozdziale. Szczegółowo omawiam metody *brute force*, Knutha-Morrisa-Pratta (KMP), Boyera i Moore’a oraz Karpa-Rabina.

Rozdział 15. „Zaawansowane techniki programowania”

Wieloletnie poszukiwania w dziedzinie algorytmiki zaowocowały wynalezieniem pewnej grupy metod o charakterze generalnym, np. programowania dynamicznego, „dziel i zwyciężaj” i algorytmów „żarłocznych” (ang. *greedy*). Zwiększają one zakres możliwych zastosowań komputerów do skuteczniejszego rozwiązywania problemów.

Rozdział 16. „Algorytmy grafowe”

Tu znajdziesz opis jednej z najciekawszych struktur danych występujących w informatyce. Grafy ułatwiają (a czasami po prostu umożliwiają) rozwiązanie wielu problemów, które traktowane za pomocą innych struktur danych wydają się nierozwiązywalne. Poznasz tu metody realizacji struktur grafowych oraz najpopularniejsze algorytmy stanowiące „cegiełki”, z których często buduje się większe systemy analizy danych.

Rozdział 17. „Matematyka i Python”

W tym rozdziale prezentuję kilka ciekawych problemów natury obliczeniowej, ukazujących zastosowanie komputerów w matematyce, a konkretnie do wykonywania obliczeń przybliżonych, jakimi są: miejsca zerowe funkcji, interpolacje, różniczkowanie, całkowanie, metoda Gaussa itp. Zapoznasz się także z podstawami wizualizacji wykresów funkcji z użyciem biblioteki Matplotlib.

Rozdział 18. „Kodowanie i kompresja danych”

Ten rozdział stanowi obszerne omówienie popularnych metod szyfrowania i kompresji danych. Zapoznasz się z pojęciem kodowania symetrycznego i asymetrycznego; omówię też szczegółowo system kryptograficzny z kluczem publicznym (RSA), przy tej okazji przedstawię także sposób wykonywania operacji arytmetycznych na bardzo dużych liczbach całkowitych. Zagadnienia kompresji danych poznasz od podstaw teoretycznych i prostych metod; opiszę tu także dokładnie słynne algorytmy kompresji metodą Huffmana i LZW.

Rozdział 19. „Czy komputery mogą myśleć...?”

Jest to wstęp do bardzo rozległej dziedziny tzw. sztucznej inteligencji. Omawiam w nim obszary zainteresowań tej dziedziny i na przykładzie gry w kółko i krzyżyk pokazuję implementację programową popularnego w teorii gier algorytmu min-max.

Dodatek A

Na końcu książki dołożyłem szczegółowe instrukcje dotyczące instalacji i używania środowiska Python i edytora PyCharm.

Konwencje typograficzne i oznaczenia

Poniżej znajduje się kilka typowych oznaczeń i konwencji, które zastosowano na kartach tej książki.

- Regułą jest, że wszystkie listingi i teksty ukazujące się na ekranie odróżniono od zasadniczej treści książki `czcionką o stałej szerokości znaków`; to samo dotyczy komend systemu operacyjnego, jeśli takie są opisywane.
- Wszelkie komendy poleceń lub wyniki działania programu w tekstowej konsoli systemowej, np. `cmd` w Windowsie lub Terminal w Linuksie albo macOS-ie, są poprzedzane symbolem dolara ($).
- Odniesienie do nazwy pliku źródłowego (np. *prog.py*) w treści książki oznacza, że pełny tekst programu znajduje się w pliku *prog.py*, umieszczonym w archiwum ZIP zawierającym przykładowe programy dostępne na serwerze FTP firmy Helion.

  Uwaga: w niektórych rozdziałach napotkasz kod wbudowany w treść akapitu bez oznaczenia nazwy pliku — zazwyczaj jest to kontynuacja opisu

rozpoczętego kilka stron wcześniej; w takiej sytuacji cofnij się o kilka stron i sprawdź, czy nazwa pliku nie została już podana. Zalecam rozpakowanie archiwum ZIP w dowolnym katalogu, co automatycznie utworzy folder o nazwie *Przykłady* zawierający skrypty, pliki robocze oraz podfoldery używane przez niektóre skrypty.

Ramka opatrzona takim symbolem zawiera proste definicje pojęć teoretycznych lub wyjaśnienia uzupełniające dotyczące wybranych elementów języka Python. Pozwoli to, jeśli zajdzie taka potrzeba, usunąć wątpliwości w kwestii mechanizmów tego języka.

Ważna uwaga — materiał istotny dla zrozumienia omawianego zagadnienia.

Ostrzeżenie — rzeczy, których nie powinieneś robić, jeśli chcesz uniknąć kłopotów.

Odwołanie — w miejscu, które wskazuje, znajdziesz dodatkowe informacje dotyczące omawianego zagadnienia.

Początki są trudne, ale przy odrobinie wytrwałości możesz zadziwiająco szybko nabyć sprawności „kodowania" do tej pory zarezerwowanej dla programistów.

Zapraszam do lektury i jak to jest tradycyjnie w przypadku moich publikacji, chciałbym Cię bardzo wyraźnie, Drogi Czytelniku, ostrzec, że wkraczasz w słynną „strefę wolną od zbędnej teorii"!

Piotr Wróblewski,
Wrocław luty – maj 2022

ROZDZIAŁ 14. Przeszukiwanie tekstów

Ten rozdział jest poświęcony metodom przeszukiwania tekstów. W informatyce jest ona słusznie traktowana jako odrębna dziedzina wiedzy z uwagi na szeroką gamę zastosowań praktycznych (systemy baz danych, kompilatory, narzędzia systemowe operujące na drzewie katalogów i listach plików…).

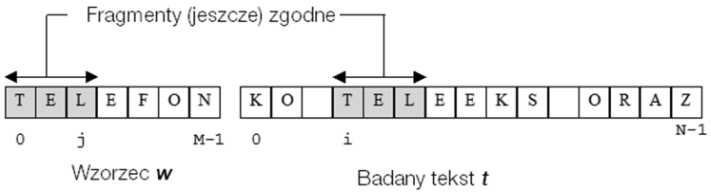
Tekst może być tutaj definiowany jako **ciąg znaków w sensie informatycznym** (nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną"!) i może być ciągiem bitów, który oczywiście można interpretować za pomocą umownych kodów (np. ASCII, Unicode) jako jednostki leksykalne mające określone znaczenie dla czytelnika. Wszystko jest zresztą kwestią umowy, w szczególności ciąg bitów może reprezentować np. obraz graficzny… (zerknij do rozdziału 2.).

Okazuje się, że przyjęcie konwencji dotyczących interpretowania informacji ułatwia wiele operacji na niej. Dlatego też pozostańmy przy ogólnikowym stwierdzeniu „tekst", wiedząc, że za tym określeniem może się kryć sporo znaczeń. Spójrzmy zatem na kilka popularnych algorytmów przeszukiwania tekstów, które każdy programista powinien znać i rozumieć.

Algorytm typu brute force

Określenie *brute force* w przypadku algorytmów zazwyczaj tłumaczymy: „na siłę" lub „siłowy", a jeden z moich znajomych pomysłowo przełożył całość jako „metodę mastodonta", co doskonale odzwierciedla jej nieco bezmyślny charakter.

Zadaniem, które będziemy usiłowali wspólnie rozwiązać, jest poszukiwanie wzorca *w* o długości *M* znaków w tekście *t* o długości *N* (ang. *pattern matching*). Z łatwością możemy zaproponować dość oczywisty algorytm rozwiązujący to zadanie, a bazować będziemy na pomysłach symbolicznie przedstawionych na rysunku 14.1.



RYSUNEK 14.1. Algorytm typu brute force przeszukiwania tekstu

Zarezerwujmy indeksy `j` i `i` do poruszania się, odpowiednio, we wzorcu i w tekście podczas operacji porównywania znak po znaku zgodności wzorca z tekstem. Załóżmy, że w trakcie poszukiwań obszary objęte szarym kolorem na rysunku okazały się zgodne. Po stwierdzeniu tego faktu przesuwamy się zarówno we wzorcu, jak i w tekście o jedną pozycję do przodu (`i++; j++`).

Cóż jednak powinno się stać z indeksami `i` oraz `j` podczas stwierdzania niezgodności znaków? W takiej sytuacji całe poszukiwanie kończy się porażką, co zmusza do anulowania „szarej strefy" zgodności. Czynimy to poprzez cofnięcie się w tekście o to, co było zgodne, czyli o `j-1` znaków, wyzerowując przy okazji `j`. Omówię jeszcze moment stwierdzenia całkowitej zgodności wzorca z tekstem. Kiedy to nastąpi? Otóż nietrudno zauważyć, że podczas stwierdzania zgodności ostatniego znaku `j` powinno się zrównać z `M`. Możemy wówczas łatwo odtworzyć pozycję, od której wzorzec startuje w badanym tekście: będzie to oczywiście `i-M`.

Tłumacząc powyższe sytuacje na instrukcje w Pythonie, możemy łatwo dojść do następującej procedury (*SzukajTXT.py*):

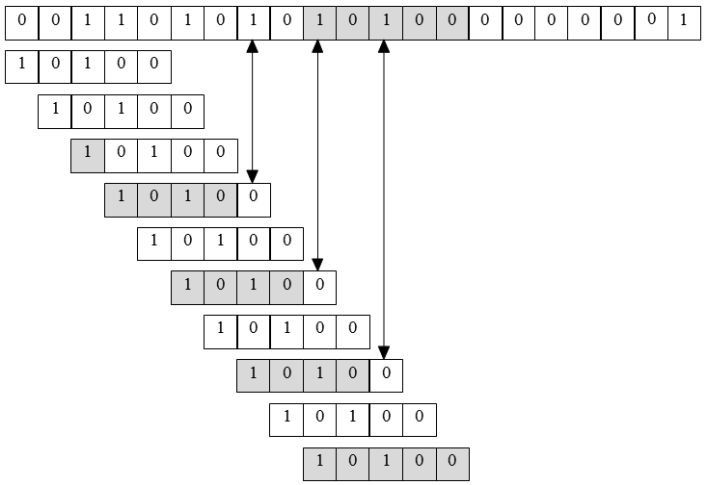
```
def szukaj(w, t):            # Konwencja: w – wzorzec, t – przeszukiwany tekst
    i, j = 0, 0
    M=len(w)
    N=len(t)
    while j<M and i<N:
        if t[i] != w[j]:     # (*) Poziome przesuwanie się wzorca
            i = i-(j-1)
            j=-1
        i=i+1                # (**)
        j=j+1
    if j==M:
        return i-M
    else:
        return -1            # Umowna porażka poszukiwania
# Testujemy:
print("Przeszukiwany ciąg znaków")
t="abrakadabra"
print(t,"\n0123456789...")
w="rak"
print(f" Szukam {w} w {t}: {szukaj(w,t)}")
w="raki"
print(f" Szukam {w} w {t}: {szukaj(w,t)}")
```

Wynik uruchomienia programu jest następujący (dodałem wyróżnienie, aby ułatwić analizę):

```
Przeszukiwany ciąg znaków
abrakadabra
0123456789...
 Szukam rak w abrakadabra: 2
 Szukam raki w abrakadabra: -1
```

Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorzec, lub `-1`, gdy poszukiwany tekst nie został odnaleziony — to znana już doskonale konwencja. Program można też nieznacznie zmodyfikować, aby wyszukiwał wszystkie wystąpienia wzorca w tekście — wystarczy w takim przypadku wypisać odnaleziony indeks i kontynuować pętlę aż do osiągnięcia końca tekstu.

Przypatrzmy się teraz dokładniej przykładowi poszukiwania wzorca `10100` w pewnym tekście binarnym (rysunek 14.2).



RYSUNEK 14.2. „Fałszywe starty" podczas poszukiwania

Rysunek jest nieco uproszczony: w istocie poziome przesuwanie się wzorca oznacza instrukcje zaznaczone na listingu *SzukajTXT.py* jako (*), natomiast cała szara strefa o długości *k* oznacza *k*-krotne wykonanie (**).

Na podstawie zobrazowanego przykładu możemy podjąć próbę wymyślenia takiego najgorszego tekstu i wzorca, dla których proces poszukiwania będzie trwał możliwie najdłużej. Chodzi oczywiście zarówno o tekst, jak i wzorzec złożone z samych zer i zakończone jedynką (umawiamy się, że zera i jedynki symbolizują tu dwa różne znaki).

Spróbujmy obliczyć **klasę tego algorytmu** dla opisanego przed chwilą ekstremalnego najgorszego przypadku. Obliczenie nie należy do skomplikowanych czynności: przy założeniu, że restart algorytmu będzie konieczny `(N-1)-(M-2)=N-M+1` razy i że podczas każdego cyklu konieczne jest wykonanie `M` porównań, otrzymujemy natychmiast `M·(N-M+1)`, czyli ok.[1] `M·N`.

Zaprezentowany w tym podrozdziale algorytm wykorzystuje komputer jako bezmyślne, ale sprawne liczydło. Jego złożoność obliczeniowa eliminuje go w praktyce z przeszukiwania tekstów binarnych, w których może wystąpić wiele niekorzystnych konfiguracji danych. Jedyną zaletą algorytmu jest jego prostota, co i tak nie czyni go wystarczająco atrakcyjnym, by dać się zamęczyć jego powolnym działaniem.

Nowe algorytmy poszukiwań

Algorytm, o którym będzie mowa w tym rozdziale, posiada ciekawą historię, którą w formie anegdoty warto przytoczyć. Otóż w 1970 r. Stephen Arthur Cook udowodnił teoretyczny rezultat dotyczący pewnej abstrakcyjnej maszyny. Wynikało z niego,

[1] Zwykle `M` będzie znacznie mniejsze niż `N`.

że istniał algorytm poszukiwania wzorca w tekście, który działał w czasie proporcjonalnym do `M+N` w najgorszym przypadku. Rezultat pracy Cooka wcale nie był przewidziany do praktycznych celów, niemniej Donald Knuth i Vaughan Ronald Pratt otrzymali na jego podstawie algorytm, który można już było zaimplementować w komputerze — ukazując przy okazji, że pomiędzy praktycznymi realizacjami a rozważaniami teoretycznymi nie istnieje wcale aż tak ogromna przepaść, jak by się mogło wydawać. W tym samym czasie James Morris odkrył dokładnie ten sam algorytm jako rozwiązanie problemu, który napotkał podczas praktycznej implementacji edytora tekstu. Algorytm KMP — bo tak będziemy go dalej zwali — jest jednym z przykładów dość częstych w nauce odkryć równoległych: z jakichś niewiadomych powodów nagle kilku pracujących osobno ludzi dochodzi do tego samego dobrego rezultatu. Prawda, że jest w tym coś niesamowitego i aż się prosi o jakieś metafizyczne hipotezy?

Knuth, Morris i Pratt opublikowali swój algorytm dopiero w 1976 r. Tymczasem pojawił się kolejny „cudowny" algorytm, tym razem autorstwa Roberta Boyera i J Strothera Moore'a, który okazał się w pewnych zastosowaniach znacznie szybszy od algorytmu KMP. Został on także równolegle wynaleziony (odkryty?) przez Billa Gospera. Oba te algorytmy są jednak dość trudne do zrozumienia bez pogłębionej analizy, co utrudniło ich rozpropagowanie.

W 1980 r. Richard Karp i Michael Rabin doszli do wniosku, że przeszukiwanie tekstów nie jest aż tak dalekie od standardowych metod przeszukiwania, i wynaleźli algorytm, który — działając ciągle w czasie proporcjonalnym do `M+N` — jest ideowo zbliżony do poznanego już algorytmu typu *brute force*. Na dodatek jest to algorytm, który można względnie łatwo uogólnić na przypadek poszukiwania w tablicach dwuwymiarowych, co sprawia, że jest potencjalnie użyteczny w obróbce obrazów.

W następnych trzech punktach szczegółowo omówię algorytmy wspomniane w tym historycznym przeglądzie.

Algorytm KMP

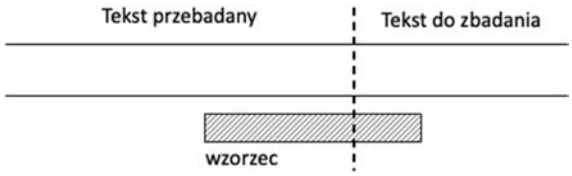
Wadą algorytmu *brute force* jest jego uwrażliwienie na konfigurację danych: fałszywe restarty są tu bardzo kosztowne; w analizie tekstu cofamy się o całą długość wzorca, zapominając po drodze o wszystkim, co przetestowaliśmy do tej pory. Narzuca się tu niejako chęć skorzystania z informacji, które już w pewien sposób posiadamy — przecież w następnym etapie będą wykonywane częściowo te same porównania co poprzednio!

W pewnych szczególnych przypadkach przy znajomości struktury analizowanego tekstu możliwe jest ulepszenie algorytmu. Jeśli przykładowo wiemy na pewno, że w poszukiwanym wzorcu pierwszy znak w ogóle się nie pojawia[2], to w razie restartu nie musimy cofać wskaźnika `i` o `j-1` pozycji, jak to było poprzednio (patrz listing *SzukajTxt.py*). W tym przypadku możemy po prostu inkrementować `i`, wiedząc,

[2] Przykład: „ABBBBBBB" — znak „A" wystąpił tylko jeden raz.

że ewentualne powtórzenie poszukiwań na pewno nic by nie dało. Owszem, można się łatwo zgodzić z twierdzeniem, że tak wyspecjalizowane teksty zdarzają się relatywnie rzadko, jednak powyższy przykład ukazuje, iż ewentualne manipulacje algorytmami poszukiwań są ciągle możliwe — wystarczy się tylko rozejrzeć. Idea algorytmu KMP polega na wstępnym zbadaniu wzorca w celu obliczenia liczby pozycji, o które należy cofnąć wskaźnik `i` w przypadku stwierdzenia niezgodności badanego tekstu ze wzorcem. Oczywiście można również rozumować w kategoriach przesuwania wzorca do przodu — rezultat będzie ten sam. To właśnie tę drugą konwencję będziemy stosować dalej. Wiemy już, że powinniśmy przesuwać się po badanym tekście nieco inteligentniej niż w poprzednim algorytmie. W przypadku zauważenia niezgodności na pewnej pozycji `j` wzorca[3] należy zmodyfikować ten indeks, wykorzystując informację zawartą w już zbadanej „szarej strefie" zgodności.

Brzmi to wszystko zapewne niesłychanie tajemniczo, pora więc jak najszybciej wyjaśnić tę sprawę, aby uniknąć możliwych nieporozumień. Popatrzmy w tym celu na rysunek 14.3.



RYSUNEK 14.3. Wyszukiwanie optymalnego przesunięcia w algorytmie KMP

Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz wzorzec bardzo wolno w prawo, patrząc jednocześnie na już zbadany tekst — tak aby obserwować ewentualne pokrycie się tej części wzorca, która znajduje się po lewej stronie przerywanej kreski, z tekstem, który jest umieszczony powyżej wzorca. W pewnym momencie może się okazać, że następuje pokrycie obu tych części. Zatrzymujemy wówczas przesuwanie i kontynuujemy testowanie (znak po znaku) zgodności obu części znajdujących się za pionową kreską.

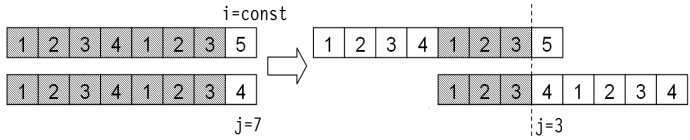
Od czego zależy ewentualne pokrycie się oglądanych fragmentów tekstu i wzorca? Otóż dość paradoksalnie badany tekst nie ma tu nic do powiedzenia — jeśli można to tak określić. Informacja o tym, jaki był, jest ukryta w stwierdzeniu „`j-1` znaków było zgodnych" — w tym sensie można zupełnie o badanym tekście zapomnieć i analizując wyłącznie wzorzec, odkryć poszukiwane optymalne przesunięcie. Na tym właśnie spostrzeżeniu opiera się idea algorytmu KMP. Okazuje się, że badając samą strukturę wzorca, można obliczyć, jak powinniśmy zmodyfikować indeks `j` w razie stwierdzenia niezgodności tekstu ze wzorcem na `j`-tej pozycji.

Zanim zagłębimy się w wyjaśnienia na temat obliczania tych przesunięć, popatrzmy na efekt ich działania na kilku kolejnych przykładach. Na rysunku 14.4 możemy

[3] Lub *i* w przypadku badanego tekstu.

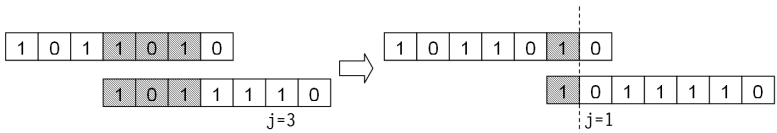
dostrzec, że na siódmej pozycji wzorca[4] (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

RYSUNEK 14.4. Przesuwanie się wzorca w algorytmie KMP (1)



Jeśli zostawimy indeks `i` w spokoju, to modyfikując wyłącznie `j`, możemy bez problemu kontynuować przeszukiwanie. Jakie jest optymalne przesunięcie wzorca? Przesuwając go wolno w prawo (rysunek 14.4), doprowadzamy w pewnym momencie do nałożenia się ciągów 123 przed kreską — cała strefa niezgodności została wyprowadzona na prawo i ewentualne dalsze testowanie może być kontynuowane!

Analogiczny przykład znajduje się na rysunku 14.5.



RYSUNEK 14.5. Przesuwanie się wzorca w algorytmie KMP (2)

Tym razem niezgodność wystąpiła na pozycji `j=3`. Wykonując podobnie jak poprzednio przesuwanie wzorca w prawo, zauważamy, że jedyne możliwe nałożenie się znaków wystąpi po przesunięciu o dwie pozycje w prawo — czyli dla `j=1`. Dodatkowo okazuje się, że znaki za kreską też się pokryły, ale o tym algorytm „dowie się" dopiero podczas kolejnego testu zgodności na pozycji `i`.

Dla algorytmu KMP konieczne okazuje się wprowadzenie tablicy przesunięć `int shift[M]`, gdzie `M` jest długością wzorca. Sposób jej zastosowania będzie następujący: jeśli na pozycji `j` wystąpiła niezgodność znaków, kolejną wartością `j` będzie `shift[j]`. Nie wnikając chwilowo w sposób inicjowania tej tablicy (odmiennej oczywiście dla każdego wzorca), możemy natychmiast podać algorytm KMP, który w konstrukcji jest niemal dokładną kopią algorytmu typu *brute force* (*KMP.py*):

```
shift=list()     # Tablica przesunięć (zmienna globalna)
def kmp(w, t):   # Konwencja: w – wzorzec, t – przeszukiwany tekst
    N=len(t)
    M=len(w)
    i, j = 0, 0
    while i<N and j<M:
        while j>=0 and t[i]!=w[j]:
            j=shift[j]
        i=i+1
        j=j+1
    if j==M:
```

[4] Licząc indeksy tablicy tradycyjnie od zera.

```
        return i-M
    else:
        return -1
```

Program można przetestować, używając prostych wywołań:

```
# Testujemy:
print("Przeszukiwany ciąg znaków")
t="abcd1010def"
print(t,"\n0123456789...")
w="1010"
init_shifts(w)
print(f" Szukam {w} w {t}: {kmp(w,t)}") # Szukam 1010 w abcd1010def: 4
w="kotek"
init_shifts(w)
print(f" Szukam {w} w {t}: {kmp(w,t)}") # Szukam kotek w abcd1010def: -1
```

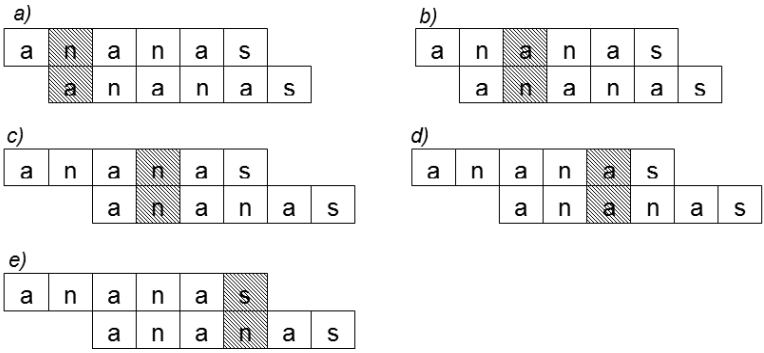
Szczególnym przypadkiem jest wystąpienie niezgodności na pozycji zerowej: z założenia niemożliwe jest tu przesuwanie wzorca w celu uzyskania nałożenia się znaków. Z tego powodu chcemy, aby indeks `j` pozostał niezmieniony, przy jednoczesnej progresji indeksu `i`. Jest to możliwe do uzyskania, jeśli umówimy się, że `shift[0]` zostanie zainicjowany wartością `-1`. Wówczas podczas kolejnej iteracji pętli `for` nastąpi inkrementacja `i` oraz `j`, co wyzeruje `j`.

Pozostaje do omówienia sposób konstrukcji tablicy `shift[M]`. Jej obliczenie powinno nastąpić *przed* wywołaniem funkcji `kmp()`, co sugeruje, że w przypadku wielokrotnego poszukiwania tego samego wzorca nie musimy już powtarzać inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna — jest ona niemal identyczna z `kmp()`, z tą tylko różnicą, że algorytm sprawdza zgodność wzorca... z nim samym!

```
def init_shifts(w):
    M = len(w)
    global shift          # Odwołanie do zmiennej globalnej
    shift = [0] * len(w)  # Inicjacja pustej tablicy przesunięć
    shift[0]=-1
    i, j = 0, -1
    while i < M-1:
        shift[i] = j
        while j>=0 and w[i]!=w[j]:
            j=shift[j]
        i = i + 1
        j = j + 1
```

Sens tego algorytmu jest następujący: tuż po inkrementacji `i` oraz `j` wiemy, że pierwsze `j` znaków wzorca jest zgodne ze znakami na pozycjach `p[i-j-1] ... p[i-1]` (ostatnie `j` pozycji w pierwszych `i` znakach wzorca). Ponieważ jest to największe `j` spełniające powyższy warunek, zatem aby nie ominąć *potencjalnego* miejsca wykrycia wzorca w tekście, należy ustawić `shift[i]` na `j`.

Popatrzmy, jaki będzie efekt zadziałania funkcji `init_shifts()` na słowie *ananas* (rysunek 14.6).



RYSUNEK 14.6. Optymalne przesunięcia wzorca „ananas" w algorytmie KMP

Zacieniowane litery oznaczają miejsca, w których wystąpiła niezgodność wzorca z tekstem. W każdym przypadku graficznie przedstawiono efekt przesunięcia wzorca — widać wyraźnie, które strefy pokrywają się przed strefą zacieniowaną (porównaj z rysunkiem 14.5).

Przypomnijmy jeszcze, że tablica `shift` zawiera nową wartość dla indeksu `j`, który przemieszcza się po wzorcu.

Algorytm KMP działa w czasie proporcjonalnym do `M+N` w najgorszym przypadku. Największy zauważalny zysk związany z jego użyciem dotyczy przypadku tekstów o wysokim stopniu samopowtarzalności — dość rzadko występujących w praktyce. Dla typowych tekstów zysk związany z wyborem metody KMP będzie zatem słabo zauważalny.

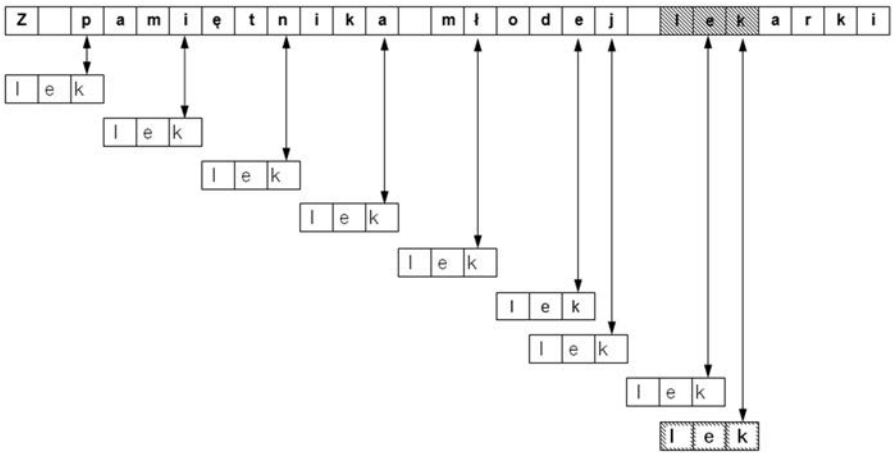
Użycie tego algorytmu jest jednak niezbędne w tych aplikacjach, w których następuje liniowe przeglądanie tekstu — bez buforowania. Jak łatwo zauważyć, wskaźnik `i` w funkcji `kmp` nigdy nie jest dekrementowany, co oznacza, że plik można przeglądać od początku do końca bez cofania się w nim. W niektórych systemach może to mieć istotne znaczenie praktyczne — przykładowo, mamy zamiar analizować bardzo długi plik tekstowy i charakter wykonywanych operacji nie pozwala na cofnięcie się w tej czynności (i w odczytywanym na bieżąco pliku).

Algorytm Boyera-Moore'a

Kolejny algorytm, który omówię, jest ideowo znacznie prostszy do zrozumienia niż algorytm KMP. W przeciwieństwie do metody KMP porównywaniu ulega ostatni znak wzorca. To niekonwencjonalne podejście ma kilka istotnych zalet:

- Jeśli podczas porównywania okaże się, że rozpatrywany aktualnie znak nie wchodzi w ogóle w skład wzorca, możemy „skoczyć" w analizie tekstu o całą długość wzorca do przodu! Ciężar algorytmu przesunął się zatem z analizy ewentualnych zgodności na badanie niezgodności, a te ostatnie są statystycznie znacznie częściej spotykane.
- Skoki wzorca są zazwyczaj znacznie większe od 1 — porównaj z metodą KMP!

Zanim przejdę do szczegółowej prezentacji kodu, omówię na przykładzie jego działanie. Spójrzmy w tym celu na rysunek 14.7, gdzie przedstawione zostało poszukiwanie ciągu znaków „lek" w tekście *Z pamiętnika młodej lekarki*[5].



RYSUNEK 14.7. Przeszukiwanie tekstu metodą Boyera-Moore'a

Pierwsze pięć porównań trafia na litery `p`, `i`, `n`, `a` i `ł`, które we wzorcu nie występują! Za każdym razem możemy zatem przeskoczyć w tekście o trzy znaki do przodu (długość wzorca). Porównanie szóste trafia jednak na literę `e`, która w słowie „lek" występuje. Algorytm wówczas przesuwa wzorzec o tyle pozycji do przodu, aby litery `e` nałożyły się na siebie, i porównywanie jest kontynuowane.

Następnie okazuje się, że litera `j` nie występuje we wzorcu — mamy zatem prawo przesunąć się o kolejne trzy znaki do przodu. W tym momencie trafiamy już na poszukiwane słowo, co następuje po jednokrotnym przesunięciu wzorca, tak aby pokryły się litery `k`.

Algorytm jest, jak widać, klarowny, prosty i szybki. Jego realizacja także nie jest zbyt skomplikowana. Podobnie jak w metodzie poprzedniej, także tu musimy wykonać pewną prekompilację w celu utworzenia tablicy przesunięć. Tym razem jednak tablica ta będzie miała tyle pozycji, ile jest znaków w alfabecie — wszystkie znaki, które mogą wystąpić w tekście, plus spacja.

Będziemy również potrzebowali prostej funkcji `indeks()`, która zwraca w przypadku spacji liczbę zero (w pozostałych przypadkach numer litery w alfabecie). Poniższy przykład uwzględnia jedynie kilka polskich liter — z łatwością można go uzupełnić o brakujące znaki. Numer litery jest oczywiście zupełnie arbitralny i zależy od programisty. Ważne jest tylko, aby nie pominąć w tablicy żadnej litery, która może wystąpić w tekście. Jedną z możliwych wersji funkcji `indeks()` przedstawiono poniżej (*BM.py*).

[5] Tytuł kultowego cyklu skeczy radiowych autorstwa Ewy Szumańskiej (1921 – 2011).

Funkcja `indeks()` ma jedynie charakter usługowy. Służy ona m.in. do właściwej inicjacji tablicy przesunięć.

```
K= 26*2 + 2*2 + 1     # Znaki ASCII + polskie litery + odstęp
shift=[0]*K           # Zmienna globalna

def indeks(c):        # Konwersja znaku 'c' na indeksy w tablicy
  if c==' ':
      return 0      )
  elif c=='ę':
      return 53
  elif c =='Ę':
      return 54
  elif c == 'ł':
      return 55
  elif c == 'Ł':
      return 56
 # itd., itd. (miejsce na kolejne polskie znaki)
  elif (c >= 'a') and (c <= 'z'):
       return ord(c) - ord('a') +1      # Od 1 do 25
  elif (c >= 'A') and (c <= 'Z'):
      return ord(c) - ord('A') + 27     # Od 27 do 52
  else:
      print("Błędny znak!")
      return -1
```

Uwaga: brak rzetelnej ochrony przed błędami — sprawdzam tylko kilka prostych warunków!

Mając za sobą analizę przykładu z rysunku 14.7, nie powinieneś być zbytnio zdziwiony sposobem inicjacji tablicy przesunięć:

```
def init_shifts(w):
    global shift
    M=len(w)
    for i in range(K):
        shift[i]=M
    for i in range(M):
        shift[ indeks(w[i]) ] = M-i-1
```

Teraz przejdziemy do prezentacji listingu algorytmu z przykładem wywołania:

```
def bm(w, t):
    global shift
    init_shifts(w)
    N = len(t)
    M = len(w)
    i, j  = M - 1, M - 1
    while j>=0:
        while t[i]!=w[j]:
            x=shift[ indeks(t[i]) ]
            if M-j>x:
                i=i+M-j
            else:
                i=i+x
            if i>=N:
```

```
                return -1
            j=M-1
        i=i-1
        j=j-1
    return i+1

# Testujemy:
t="Z pamiętnika młodej lekarki"
#  0123456789012345678901234567890
print(t)
print("012345678901234567890123456789")
print(f"Wynik/pozycja poszukiwań słowa 'lek'={bm('lek', t)}")          # Wynik: 20
print(f"Wynik/pozycja poszukiwań słowa 'pa'={bm('pa', t)}")            # Wynik: 2
print(f"Wynik/pozycja poszukiwań słowa 'parapet'=, {bm('parapet', t)}") # Wynik: -1
print(f"Wynik/pozycja poszukiwań słowa 'młodej'={bm('młodej', t)}")    # Wynik: 13
print(f"Wynik/pozycja poszukiwań słowa 'koperta'= {bm('koperta', t)}") # Wynik: -1
```

Algorytm Boyera-Moore'a, podobnie jak KMP, jest klasy `M+N`, jednak jest o tyle od niego lepszy, że w przypadku krótkich wzorców i długiego alfabetu kończy się po ok. `M/N` porównaniach. W celu obliczenia optymalnych przesunięć[6] autorzy algorytmu proponują połączenie powyższego algorytmu z zaproponowanym przez Knutha, Morrisa i Pratta. Celowość tego zabiegu wydaje się jednak wątpliwa, gdyż optymalizując sam algorytm, można bardzo łatwo sprawić, że proces prekompilacji wzorca stanie się zbyt czasochłonny.

Algorytm Rabina-Karpa

Ostatni algorytm do przeszukiwania tekstów, który przeanalizuję, wymaga znajomości rozdziału 11. i terminologii transformacji kluczowej, która została w nim przedstawiona.

Algorytm Rabina-Karpa polega na dość przewrotnej idei:

- Wzorzec `w` (do odszukania) jest *kluczem* o długości `M` znaków, charakteryzującym się pewną wartością wybranej przez nas funkcji `H`. Możemy zatem obliczyć jednokrotnie H_w=`H(w)` i korzystać z tego wyliczenia w sposób ciągły.
- Tekst wejściowy `t` (do przeszukania) może być odczytywany w taki sposób, aby na bieżąco znać `M` ostatnich znaków[7]. Z tych `M` znaków wyliczamy na bieżąco H_t=`H(t)`.

Gdy założymy jednoznaczność wybranej funkcji `H`, sprawdzenie zgodności wzorca z aktualnie badanym fragmentem tekstu sprowadza się do odpowiedzi na pytanie: czy H_w jest równe H_t? Jeśli jesteś spostrzegawczy, masz prawo pokręcić w tym miejscu z powątpiewaniem głową i stwierdzić, że przecież to nie ma prawa działać szybko! Istotnie pomysł wyliczenia dodatkowo funkcji `H` dla każdego słowa wejściowego o długości `M` wydaje się tak samo kosztowny — jeśli nie bardziej — jak zwykłe sprawdzanie tekstu znak po znaku (np. stosując algorytm siłowy typu

[6] Rozważ np. wielokrotne występowanie takich samych liter we wzorcu.

[7] Na samym początku będzie to oczywiście `M` pierwszych znaków tekstu.

brute force). Tym bardziej że do tej pory nie powiedziałem ani słowa na temat funkcji `H`! W rozdziale 11. mogliśmy się przekonać, że jej wybór wcale nie jest taki oczywisty.

Omawiany algorytm jednak istnieje i na dodatek działa szybko! Aby zatem to wszystko, co poprzednio zostało napisane, logicznie się łączyło, potrzebny będzie jakiś trik. Sztuka polega na właściwym wyborze funkcji `H`. Robin i Karp wybrali taką funkcję, która dzięki swym szczególnym właściwościom umożliwia dynamiczne wykorzystywanie wyników obliczeń wykonanych krok wcześniej, co znacząco może uprościć obliczenia wykonywane w kroku bieżącym.

Załóżmy, że ciąg `M` znaków będziemy interpretować jako pewną liczbę całkowitą. Przyjmując za `b` jako podstawę systemu liczbę wszystkich możliwych znaków, otrzymamy:

$$x = t[i]b^{M-1} + t[i+1]b^{M-2} + \cdots + t[i+M-1].$$

Przesuńmy się teraz w tekście o jedną pozycję do przodu i zobaczmy, jak zmieni się wartość `x`:

$$x' = t[i+1]b^{M-1} + t[i+2]b^{M-2} + \cdots + t[i+M].$$

Jeśli dobrze przyjrzysz się `x` i `x'`, zobaczysz, że wartość `x'` jest w dużej części zbudowana z elementów tworzących `x` pomnożonych przez `b` z uwagi na przesunięcie. Nietrudno wówczas wywnioskować, że:

$$x' = (x - t[i]b^{M-1}) + t[i+M].$$

Jako funkcji `H` użyjemy klasycznej realizacji `H(x)=x%p`, gdzie `p` jest dużą liczbą pierwszą. Załóżmy, że dla danej pozycji `i` wartość `H(x)` jest znana. Po przesunięciu się w tekście o jedną pozycję w prawo pojawia się konieczność wyliczenia wartości funkcji `H(x')` dla tego „nowego" słowa. Czy faktycznie trzeba powtarzać całe wyliczenie? Być może istnieje pewne ułatwienie bazujące na zależności, jaka istnieje między `x` i `x'`?

Z pomocą przychodzi tu własność funkcji `modulo` użytej w wyrażeniu arytmetycznym. Można oczywiście obliczyć `modulo` z wyniku końcowego, lecz to bywa czasami niewygodne, np. z uwagi na wielkość liczby, z którą mamy do czynienia, a poza tym gdzie tu byłby zysk szybkości?! Jednak identyczny wynik otrzymuje się, aplikując funkcję `modulo` po każdej operacji cząstkowej i przenosząc otrzymaną wartość do następnego wyrażenia cząstkowego! Dla przykładu weźmy obliczenie:

$$(5 \cdot 100 + 6 \cdot 100 + 8)\%7 = 568\%7 = 1.$$

Wynik ten jest oczywiście prawdziwy, co można łatwo sprawdzić na kalkulatorze. Identyczny rezultat da jednak następująca sekwencja obliczeń:

$$(5 \cdot 100)\%7 = 3 \quad \rightarrow \quad (\mathbf{3} + 6 \cdot 100)\%7 = 0 \quad \rightarrow \quad (\mathbf{0} + 8)\%7 = 1,$$

co jest też łatwe do weryfikacji.

Implementacja algorytmu jest prosta, lecz zawiera kilka instrukcji wartych omówienia (pomijam wydruk funkcji `index()`, który jest identyczny jak na listingu *BM.py*).

Spójrz na realizację algorytmu (listing *RK.py*):

```
p=33554393          # Duża liczba pierwsza
MAX = 64            # Liczba znaków alfabetu
def rk(w, t):
    Hw = 0          # Funkcja H dla wzorca
    Ht = 0          # Funkcja H dla tekstu
    M=len(w)
    N=len(t)

    bM_1 = 1
    for i in range(1, M): # Wyliczymy wartość pow(MAX, M-1) % p
        bM_1= (MAX*bM_1) % p

    for i in range(M):
        Hw = (Hw * MAX + indeks( w[i]) ) % p   # Inicjacja funkcji H dla wzorca
        Ht = (Ht * MAX + indeks( t[i]) ) % p   # Inicjacja funkcji H dla tekstu

    i=0
    while Hw != Ht:         # Przesuwanie się w tekście
        if i+M >= N:
            return -1       # Niepowodzenie poszukiwań
        else:
            Ht = (Ht+MAX * p - indeks(t[i]   ) * bM_1) % p # (*)
            Ht = (Ht * MAX   + indeks(t[i+M])) % p
        i=i+1
    return i
```

Na pierwszym etapie następuje wyliczenie początkowych wartości `Ht` i `Hw`. Nie dotyczy to jednak aktualnie badanego fragmentu tekstu — tutaj wartość `Ht` ulega zmianie podczas każdej inkrementacji zmiennej `i`. Do obliczenia `H(x')` możemy wykorzystać omówioną wcześniej własność funkcji `modulo`. Dodatkowego wyjaśnienia wymaga być może linia listingu oznaczona (*), pozwalająca uniknąć przypadkowego wskoczenia w liczby ujemne. Gdyby istotnie tak się stało, przeniesiona do następnego wyrażenia arytmetycznego wartość `modulo` byłaby nieprawidłowa i sfałszowałaby końcowy wynik!

Kolejne uwagi dotyczą parametrów `p` i `MAX`. Zaleca się, aby `p` było dużą liczbą pierwszą[8], jednakże nie można tu przesadzać z uwagi na możliwe przekroczenie zakresu pojemności użytych zmiennych. W przypadku wyboru dużego `p` zmniejszamy prawdopodobieństwo wystąpienia kolizji spowodowanej niejednoznacznością funkcji `H`. Możliwość ta, mimo że mało prawdopodobna, ciągle istnieje i dlatego po upewnieniu się, że `Hw == Ht` ostrożny programista powinien sprawdzać już klasycznie (znak po znaku), czy wyszukanie się powiodło (dla zwróconego indeksu `i` należy porównać zawartość `w` oraz ciągu `t[i]… t[i+M-1]`).

[8] W naszym przypadku jest to liczba 33 554 393.

Co zaś się tyczy wyboru podstawy systemu (oznaczonej w programie jako `MAX`), warto wybrać liczbę nawet nieco za dużą, zawsze jednak będącą potęgą liczby 2. Możliwe jest wówczas zaimplementowanie operacji mnożenia przez `MAX` jako przesunięcia bitowego — wykonywanego przez komputer znacznie szybciej niż zwykłe mnożenie. Przykładowo dla `M = 64` możemy zapisać mnożenie `MAX*p` jako `p<<6`.

Gwoli formalności można jeszcze dodać, że gdy nie występuje kolizja (typowy przypadek!), algorytm Robina-Karpa wykonuje się w czasie proporcjonalnym do `M+N`.

Wynik uruchomienia programu jest identyczny, jak poprzednio, dla algorytmu Boyera-Moore'a.

Kilka prostych zadań

ZADANIE 1

Napisz program sprawdzający, czy zdanie wejściowe jest palindromem (tzn. czy da się czytać tak samo z lewej do prawej, jak i z prawej do lewej strony). Przykład takiego zdania: „kobyła ma mały bok".

ZADANIE 2

Napisz program sprawdzający, czy dwa zdania podane jako parametry są anagramami (tzn. zawierają wszystkie swoje znaki). Dla uproszczenia załóżmy, że zdania podajemy bez dużych liter.

ZADANIE 3

Napisz program sprawdzający, czy zdanie wejściowe zawiera ukryte słowo. Przykładowo, tekst „**Br**onek **a**lergicznie nie znosił **m**akaronu z **ka**szą" ukrywa słowo „Bramka".

Rozwiązania

Zadanie 1.

Zadanie należy do elementarnych, nie powinno zatem nikomu sprawić trudności dojście do następującego (lub zbliżonego do niego) rozwiązania (*Palindromy.py*):

```
def palindrom(s):
    dl = len (s)
    cpt = 0
    test = True         # 's' jest (na razie) palindromem
    while cpt <= dl // 2 and test == True:
        if s[cpt] == s[dl-cpt-1]:
            cpt = cpt+1
        else:
            test = False
    print(s, end="")
    if test == True:
        print(" ...jest palindromem")
```

```
    else:
        print(" ...jest zwykłym słowem...")
# -----------------------------------------
palindrom("ab")       # ab... jest zwykłym słowem
palindrom("a")        # a... jest palindromem
palindrom("abba")     # abba... jest palindromem
palindrom("abkba")    # abkba... jest palindromem
palindrom("abkca")    # abkca... jest zwykłym słowem...
```

Zadanie 2.

To zadanie wydaje się pozornie skomplikowane, ale jeśli wykorzystamy kilka sztuczek Pythona polegających na manipulowaniu zawartością listy znaków składających się na napis, to nagle wszystko stanie się bardzo proste! Cała sztuka polega na konwersji napisów na listy znaków i przetworzeniu nowo utworzonych napisów (ich kopii bez spacji) na tablice znaków. Po posortowaniu tablic wynikowych możemy je porównać, sprawdzając, czy mamy do czynienia z anagramami (*Anagramy.py*):

```
def anagramy(zdanie1, zdanie2):
    print("s1=", zdanie2, ", s2=", zdanie1, end=", wynik: ")
    s1 = list(zdanie1)                      # Zamiana na listę znaków
    s2 = list(zdanie2)                      # jw.
    s1 = [e for e in s1 if e not in [' '] ] # Usuwamy spacje
    s2 = [e for e in s2 if e not in [' '] ] # jw.
    s1=sorted(s1)                           # Sortujemy znaki w s1
    s2=sorted(s2)                           # Sortujemy znaki w s2

    if s1==s2:
        print(", \t --> jest anagramem")
    else:
        print(", \t --> nie jest anagramem")

# Testujemy:
# Klasyczny przykład to pytanie Piłata: „Co to jest prawda?" i odpowiedź Jezusa: „Człowiek, który
# stoi przed tobą":
anagramy ("quid est veritas","vir est qui adest") # Jest anagramem
anagramy ("baba","abba")                          # Jest anagramem
anagramy ("baba","abbe")                          # Nie jest anagramem
```

Zadanie 3.

W rozwiązaniu zadania ponownie wykorzystamy konwersję napisów na listy znaków i spróbujemy usunąć **wszystkie** znaki zawarte w liście reprezentującej poszukiwane z listy znaków będącej odpowiednikiem zdania. Po wykonaniu tego zadania proste porównanie długości list odpowie nam na pytanie! Popatrz na listing *Bronek.py*:

```
def zawiera(wyraz, zdanie):
    print(f"Szukam '{wyraz}' w '{zdanie}'")
    n = len(wyraz)
    if n > len(zdanie):
        print("Poszukiwane słowo jest dłuższe niż zdanie...")
        return False
```

```
    wyraz_lst = list(wyraz)                    # Zamiana na listę znaków
    zdanie_lst = list(zdanie)                  # jw.
    print(len(zdanie_lst), "  ", zdanie_lst)   # Usuwamy znaki zawarte w wyraz_lst
                                               # ze zdania
    for e in wyraz_lst:
        try:
            zdanie_lst.remove(e)
        except ValueError:
            pass                               # Nic nie robimy, gdy elementu nie ma
    print(len(zdanie_lst), "  ", zdanie_lst)   # Wydruk kontrolny
    print(len(wyraz_lst), "  ", wyraz_lst)    # Wydruk kontrolny
    return len(zdanie) - n == len(zdanie_lst)

# Testujemy:
print ( zawiera("Bramka","Bronek alergicznie nie znosił makaronu z kaszą") )
print ( zawiera("Kia","Salon samochodów Renault i Nissan") )
```

Przykładowe wyniki:

```
Szukam 'Bramka' w 'Bronek alergicznie nie znosił makaronu z kaszą'
46    ['B', 'r', 'o', 'n', 'e', 'k', ' ', 'a', 'l', 'e', 'r', 'g', 'i', 'c', 'z',
➥'n', 'i', 'e', ' ', 'n', 'i', 'e', ' ', 'z', 'n', 'o', 's', 'i', 'ł', ' ', 'm',
➥'a', 'k', 'a', 'r', 'o', 'n', 'u', ' ', 'z', ' ', 'k', 'a', 's', 'z', 'ą']
40    ['o', 'n', 'e', ' ', 'l', 'e', 'r', 'g', 'i', 'c', 'z', 'n', 'i', 'e', ' ',
➥'n', 'i', 'e', ' ', 'z', 'n', 'o', 's', 'i', 'ł', ' ', 'k', 'a', 'r', 'o', 'n',
➥'u', ' ', 'z', ' ', 'k', 'a', 's', 'z', 'ą']
6    ['B', 'r', 'a', 'm', 'k', 'a']
True
Szukam 'Kia' w 'Salon samochodów Renault i Nissan'
33    ['S', 'a', 'l', 'o', 'n', ' ', 's', 'a', 'm', 'o', 'c', 'h', 'o', 'd', 'ó',
➥'w', ' ', 'R', 'e', 'n', 'a', 'u', 'l', 't', ' ', 'i', ' ', 'N', 'i', 's', 's',
➥'a', 'n']
31    ['S', 'l', 'o', 'n', ' ', 's', 'a', 'm', 'o', 'c', 'h', 'o', 'd', 'ó', 'w',
➥' ', 'R', 'e', 'n', 'a', 'u', 'l', 't', ' ', ' ', 'N', 'i', 's', 's', 'a', 'n']
3    ['K', 'i', 'a']
False
```

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Python? Idealny dla praktyków!

Wiernym czytelnikom publikacji spod znaku wydawnictwa Helion Piotra Wróblewskiego przedstawiać nie trzeba. Dość wspomnieć, że jest on autorem wielu publikacji poświęconych głównie programowaniu i obsłudze komputerów. Jego najnowsza książka, *Algorytmy w Pythonie. Techniki programowania dla praktyków*, to ponad 500 stron konkretnych informacji związanych z nauką programowania w Pythonie.

Podręcznik jest przeznaczony przede wszystkim dla tych, którzy poszukują prostego i praktycznego samouczka. Z powodzeniem jednak skorzystają z niego również osoby chcące się znaleźć na wyższym — bardziej świadomym — poziomie programowania, na którym pisanie kodu nie odbywa się już metodą prób i błędów.

Konwencja przyjęta przez autora opiera się na zasadzie „minimum teorii, maksimum praktyki". Pracę z podręcznikiem ułatwiają liczne zadania, definicje, listingi, uwagi, rysunki, tabele i ostrzeżenia; nie brak tu także humoru. Porady dotyczące instalacji i korzystania ze środowiska Pythona pozwalają szybko przejść od teorii do praktyki. Podane kody źródłowe programów zaś są gotowe do uruchomienia i zostały przetestowane w najnowszej edycji Pythona działającego pod kontrolą systemów operacyjnych Windows, macOS i Linux.

Na koniec ciekawostka. Podręcznik rozpoczynają dwa haiku Dariusza Brzóski-Brzóskiewicza, gdańskiego performera i poety, co samo w sobie stanowi nietuzinkową zapowiedź książki na temat programowania.

- **Systemy obliczeniowe bez tajemnic**
- **Typy proste i złożone oferowane przez Pythona**
- **Rekurencja nie boli, a nawet pomaga!**
- **Modelowanie abstrakcyjnych struktur danych**
- **Przykładowe realizacje wybranych struktur danych**
- **Struktury danych o dostępie ograniczonym**
- **Drzewa i ich reprezentacje**
- **Algorytmy przeszukiwania, sortowania, grafowe**
- **Derekursywacja i optymalizacja algorytmów**
- **Przeszukiwanie tekstów**
- **Zastosowania tablic NumPy i biblioteki Matplotlib**
- **Zaawansowane techniki programowania**
- **Kodowanie i kompresja danych**

Programowanie w Pythonie — praktycznie i z poczuciem humoru!

Piotr Wróblewski — doświadczony kierownik produktu. Pracuje w dużej firmie telekomunikacyjnej, wcześniej był także liderem zespołu testerów oprogramowania i kierownikiem projektów informatycznych. Od 1992 roku stały współpracownik wydawnictwa Helion. Autor wielu cenionych książek.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-9368-4

9 788328 393684

Cena: 119,00 zł