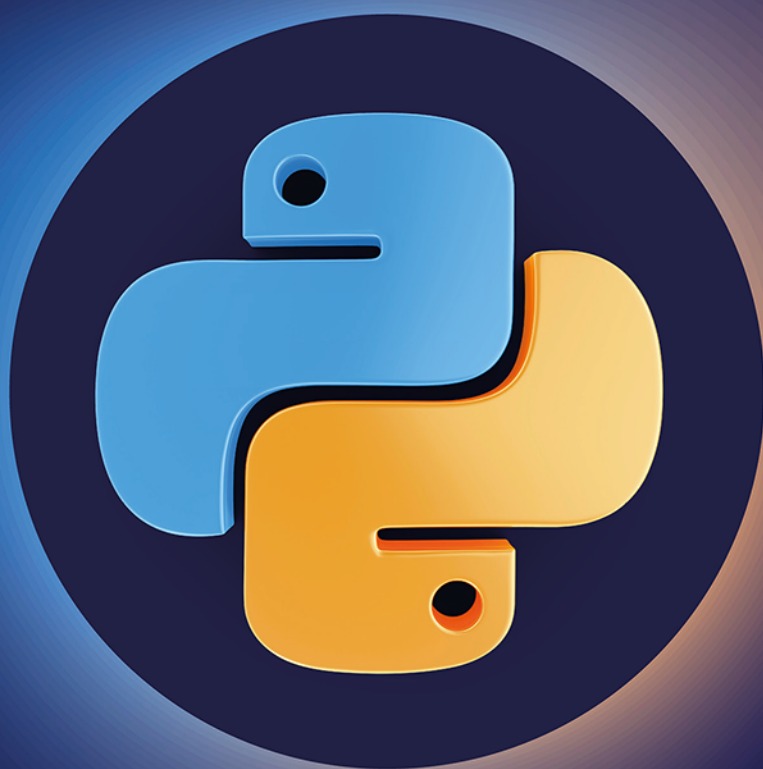


Piotr Wróblewski

Algorytmy w Pythonie

TECHNIKI PROGRAMOWANIA
DLA PRAKTYKÓW



Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

Copyright © Helion S.A. 2022

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	13
Przykładowe programy w Pythonie	14
Co odróżnia tę książkę od innych podręczników?	15
Jak należy czytać tę książkę?	16
Co zostało opisane w tej książce?	17
Konwencje typograficzne i oznaczenia	20
ROZDZIAŁ 1. Zanim wystartujemy	23
Czym powinien się charakteryzować algorytm?	24
Jak to wcześniej bywało, czyli wyjątki z historii maszyn algorytmicznych	26
Jak to się niedawno odbyło, czyli o tym, kto wymyślił metodologię programowania	31
Proces koncepcji programów	32
Poziomy abstrakcji opisu i wybór języka	33
Maszyna Turinga	35
Modelowanie i realizacja algorytmów	37
Schematy blokowe	38
Przetwarzanie danych — operatory	40
Grupowanie fragmentów kodu w funkcje	40
Pobieranie lub wyświetlanie wartości	41
Iteracyjne wykonywanie kodu	42
Operatory logiczne	45
Poprawność algorytmów	47
Zadania	49
Rozwiązania i wskazówki do zadań	50
ROZDZIAŁ 2. Systemy obliczeniowe i podstawy kodowania	53
Systemy pozycyjne w pigułce	53
System dwójkowy	55
System szesnastkowy	56
System ósemkowy	57
Operacje arytmetyczne na liczbach dwójkowych	58
Kod BCD	60
Kodowanie liczb ze znakiem	62
Kod znak-moduł (ZM)	62
Kod U2 (system uzupełnienia dwójkowego)	63
Zmienne w pamięci komputera	64
Kodowanie znaków	65
Kodowanie obrazów	68
Mapy bitowe na przykładzie formatu BMP	69

ROZDZIAŁ 3. Rekurencja	75
Definicja rekurencji	75
Ilustracja pojęcia rekurencji	77
Jak wykonują się programy rekurencyjne?	78
Niebezpieczeństwa rekurencji	80
Ciąg Fibonacciego	81
Stack overflow!	83
Pułapek ciąg dalszy	85
Stąd do wieczności	85
Definicja poprawna, ale...	86
Typy programów rekurencyjnych	88
Myślenie rekurencyjne	90
Przykład 1. Spirala	90
Przykład 2. Kwadraty „parzyste”	94
Przeszukiwanie binarne	96
Uwagi praktyczne na temat technik rekurencyjnych	98
Zadania	99
Rozwiązania i wskazówki do zadań	101
ROZDZIAŁ 4. Analiza złożoności algorytmów	105
Definicje i przykłady	106
Jeszcze raz funkcja silnia	110
Wyszukiwanie wspólnego elementu w kolekcjach	111
Zerowanie fragmentu tablicy	116
Wpadamy w pułapkę	117
Różne typy złożoności obliczeniowej	118
Nowe zadanie: uprościć obliczenia!	120
Analiza programów rekurencyjnych	121
Terminologia i definicje	121
Ilustracja metody na przykładzie	123
Rozkład logarytmiczny	124
Przeszukiwanie binarne...	
tym razem bez matematyki wyższej!	126
Zamiana dziedziny równania rekurencyjnego	127
Funkcja Ackermanna, czyli coś dla smakoszy	127
Złożoność obliczeniowa to nie religia!	129
Techniki optymalizacji programów	130
Trochę praktyki: pomiary czasów wykonania	131
Zadania	132
Rozwiązania i wskazówki do zadań	133
ROZDZIAŁ 5. Typy proste wbudowane	135
Zmienne (nie zawsze) proste	136
Typy proste	137
Stałe symboliczne	138
Pojęcie referencji	139
Zasięg zmiennych	140

Napis niezmienny jest i basta!	141
Konwersje napisów na liczby (i odwrotnie)	142
Formatowanie wyników z użyciem notacji f''	143
Tablice (nie całkiem) klasyczne	147
Inicjalizacja tablic o stałym wymiarze	149
Tuple (czasem zwane krotkami)	150
Modyfikacja tupli	151
Zastosowania programistyczne	152
ROZDZIAŁ 6. Modelowanie abstrakcyjnych struktur danych	155
Szablon nowej struktury danych	156
Dokładamy logikę biznesową	161
Błędne użycie obiektów klasy i wyjątki	165
Przeciążanie operatorów arytmetycznych	166
Poszerzanie definicji modelu danych	168
Rekurencyjne struktury danych	171
Pułapki OOP w Pythonie	172
ROZDZIAŁ 7. Przykładowe realizacje wybranych struktur danych	173
Listy jednokierunkowe	175
„Tablicowa” implementacja list	200
Listy innych typów	206
Listy z iteratorem	210
Zbiory	213
Modelowanie kolekcji — podsumowanie	216
ROZDZIAŁ 8. Typy złożone wbudowane	217
Listy, czyli tablice dynamiczne	218
Metody dostępne dla list w Pythonie	221
Listy tworzone na podstawie wyrażeń	223
Zbiory	223
Zbiory tworzone na podstawie wyrażeń	228
Słowniki	228
Szybkie tablice NumPy	234
Instalacja	235
N-wymiarowe tablice NumPy	235
Tablice i macierze NumPy	236
Deklarowanie tablic i macierzy NumPy	237
Funkcje tablicowe NumPy	239
Zmiany układu i rozmiaru tablic NumPy	242
Wycinki w tablicach	244
ROZDZIAŁ 9. Struktury danych o dostępie ograniczonym	247
Stos	247
Zasada działania stosu	247
Realizacja programowa stosu	249
Sprawdzanie typu danych obiektu	251

Kolejki FIFO	251
Sterty i kolejki priorytetowe	254
Sortowanie za pomocą sterty	262
ROZDZIAŁ 10. Drzewa i ich reprezentacje	263
Binarne drzewa poszukiwań (BST)	268
Drzewa binarne i wyrażenia arytmetyczne	273
Uniwersalna struktura słownikowa	279
Zajętość pamięci słownika	285
Drzewa „egzotyczne”	285
ROZDZIAŁ 11. Algorytmy przeszukiwania	287
Przeszukiwanie liniowe	287
Generyczne funkcje porównawcze	288
Przeszukiwanie binarne	289
Transformacja kluczowa (hashing)	291
W poszukiwaniu funkcji $H()$	293
Najbardziej znane funkcje $H()$	294
Obsługa konfliktów dostępu	297
Powrót do źródeł	297
Jeszcze raz tablice!	298
Próbkowanie liniowe	299
Podwójne kluczowanie	301
Zastosowania transformacji kluczowej	303
Klasyczne funkcje hashujące	303
Piszemy własną tablicę hashującą	304
Podsumowanie metod transformacji kluczowej	308
Przeszukiwanie danych w wybranych strukturach Pythona	309
ROZDZIAŁ 12. Algorytmy sortowania	311
Sortowanie przez wstawianie, algorytm klasy $O(N^2)$	312
Sortowanie bąbelkowe, algorytm klasy $O(N^2)$	314
Sortowanie szybkie (Quicksort) — algorytm klasy $O(N \log N)$	316
Scalanie zbiorów posortowanych	320
Sortowanie przez scalanie, algorytm klasy $O(N \log N)$	321
Sortowanie zewnętrzne	322
Sortowanie z użyciem bibliotek Pythona	326
Uwagi praktyczne	328
ROZDZIAŁ 13. Derekursywacja i optymalizacja algorytmów	329
Jak pracuje kompilator?	330
Odrobina formalizmu nie zaszkodzi!	332
Kilka przykładów derekursywacji algorytmów	333
Derekursywacja z wykorzystaniem stosu	337
Eliminacja zmiennych lokalnych	338
Metoda funkcji przeciwnych	340

Klasyczne schematy derekursywacji	342
Schemat typu while	343
Schemat typu if-else	344
Schemat z podwójnym wywołaniem rekurencyjnym	346
Podsumowanie	348
ROZDZIAŁ 14. Przeszukiwanie tekstów	349
Algorytm typu brute force	349
Nowe algorytmy poszukiwań	351
Algorytm KMP	352
Algorytm Boyera-Moore'a	356
Algorytm Rabina-Karpa	359
Kilka prostych zadań	362
Rozwiązania	362
ROZDZIAŁ 15. Zaawansowane techniki programowania	365
Programowanie typu „dziel i zwyciężaj”	366
Znajdowanie minimum i maksimum w tablicy liczb	367
Mnożenie macierzy o rozmiarze $N \times N$	370
Mnożenie liczb całkowitych	373
Inne znane algorytmy „dziel i zwyciężaj”	374
Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas...	374
Problem plecakowy, czyli niełatwe jest życie turysty piechura	375
Wydawanie reszty, czyli „A nie ma pan drobnych?” w praktyce	378
Programowanie dynamiczne	379
Ciąg Fibonacciego	381
Równania z wieloma zmiennymi	383
Najdłuższa wspólna podsekwencja	385
Najdłuższy wspólny podłańcuch	387
Heurystyki i inne techniki programowania	389
Uwagi bibliograficzne	392
ROZDZIAŁ 16. Algorytmy grafowe	393
Definicje i pojęcia podstawowe	395
Etykiety i wartości	396
Cykle w grafach	398
Sposoby reprezentacji grafów	401
Reprezentacja tablicowa	401
Słowniki węzłów	404
Listy kontra zbiory	405
Podstawowe operacje na grafach	406
Suma grafów	406
Kompozycja grafów	406
Graf do potęgi	407
Algorytm Warshalla	409
Algorytm Floyd-Warshalla	413
Algorytm Dijkstry	416
Algorytm Bellmana-Forda	418

Drzewo rozpinające minimalne	419
Algorytm Kruskala	419
Algorytm Prima	420
Przeszukiwanie grafów	421
Strategia „w głąb” (przeszukiwanie zstępujące)	422
Strategia „wszerz”	424
Inne strategie przeszukiwania	426
Problem właściwego doboru	426
Podsumowanie	431
ROZDZIAŁ 17. Matematyka i Python	433
Biblioteki naukowe dla Pythona	434
Klasyczne funkcje i metody matematyczne	434
Funkcje matematyczne NumPy	436
Poszukiwanie miejsc zerowych funkcji	437
Iteracyjne obliczanie wartości funkcji	438
Interpolacja funkcji metodą Lagrange’a	439
Różniczkowanie funkcji	441
Całkowanie funkcji metodą Simpsona	443
Rozwiązywanie układów równań liniowych metodą Gaussa	444
Uwagi końcowe	446
Wizualizacja danych z użyciem Matplotlib	447
Instalacja pakietu	447
Pierwszy wykres	448
Modyfikacje wyglądu wykresu	449
Wykresy statystyczne	452
ROZDZIAŁ 18. Kodowanie i kompresja danych	455
Kodowanie danych i arytmetyka dużych liczb	458
Metody prymitywne	458
Kodowanie symetryczne	461
Kodowanie asymetryczne	463
Kodowanie Base64	465
Obliczenia na bardzo dużych liczbach całkowitych	467
Reprezentacja dużych liczb całkowitych	467
Wyliczanie wartości modulo	470
Wybrane techniki łamania kodów	471
Jakość klucza szyfrującego	471
Metody łamania szyfrów	471
Techniki kompresji danych	473
Kompresja za pomocą modelowania matematycznego	475
Kompresja metodą RLE	476
Kompresja danych metodą Huffmana	477
Kodowanie LZW	483

ROZDZIAŁ 19. Czy komputery mogą myśleć...?	491
Przegląd obszarów zainteresowań sztucznej inteligencji (SI)	492
Systemy eksperckie	493
Sieci neuronowe	495
Reprezentacja problemów	497
Gry dwuosobowe i drzewa gier	500
Algorytm min-max	502
DODATEK A. Python — lokalne środowisko pracy	513
Testujemy poprawność instalacji Pythona	515
Instalator pip i biblioteki Pythona	518
Edytory do Pythona	519
Środowiska IDE (i dlaczego PyCharm)	521
IDLE	521
PyCharm	525
Visual Studio Community	530
Dokumentacja Pythona	531
Używanie zasobów GitHuba	533
Literatura	535
Spis rysunków	537
Spis tabel	540
Skorowidz	542



Lepiej

Lepiej bez przerwy

Iść po grudzie?

Czy

Ciągle pod górę?

Himalaje

Każdy

Ma

Swoje

Himalaje

Dariusz Brzóska-Brzósiewicz

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, nieradko i na pewno zbyt szybko przechodzi od pomysłu do realizacji, wpadając w cykl, który na początku książki nieco ironicznie zilustrowałem cytatami z haiku Dariusza Brzóska-Brzósiewiczza — 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 wszystkim 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 nieznaną, 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 skowidzem.



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 przejrzania 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 utrwała 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 wyrażań 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 arcydzieł 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 obszernie 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.).

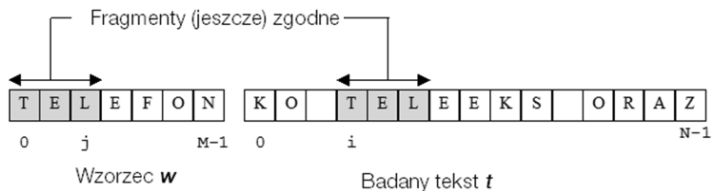
Okazuje się, że przyjęcie konwencji dotyczących interpretowania informacji ułatwia wiele operacji na niej. Dlatego też pozostaniemy przy ogólnikowym stwierdzeniu „tekst”, wiedząc, że za tym określeniem może się kryć sporo znaczeń. Spójrzmy zatem na kilka popularnych algorytmów przeszukiwania tekstów, 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 usiłovali 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 bazywać będziemy na pomysłach symbolicznie przedstawionych na rysunku 14.1.

RYСУNEK 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 wzorec 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 - wzorec, 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)}")
```

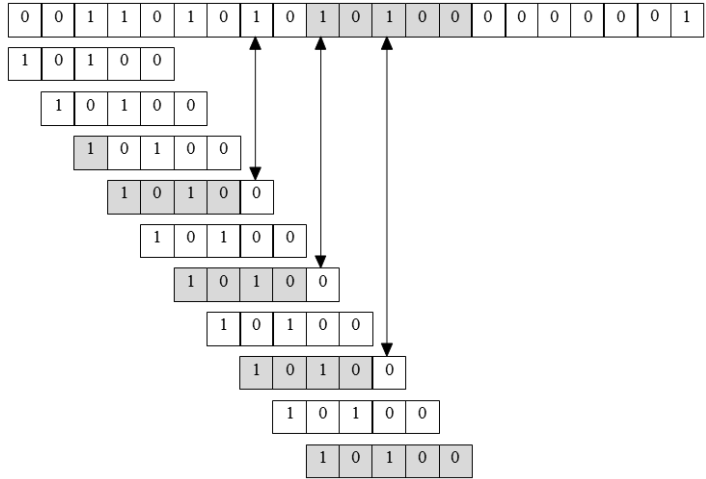
Wynik uruchomienia programu jest następujący (dodałem wyróżnienie, aby ułatwić analizę):

```
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 wzorec, 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).

RYСУNEK 14.2.
„Falszywe 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 wzorec złożone z samych zer i zakończone jedyneką (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 $M \cdot (N-M+1)$, czyli ok.¹ $M \cdot N$.

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 równoległe 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 pojawia², 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,

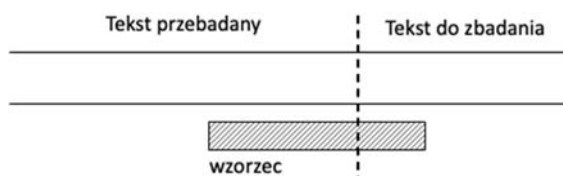
² Przykład: „ABBBBBBB” — znak „A” wystąpił tylko jeden raz.

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 przesuwac się po badanym tekście nieco inteligentniej niż w poprzednim algorytmie. W przypadku zauważenia niezgodności na pewnej pozycji j wzorca³ 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.

RYСУNEK 14.3.

Wyszukiwanie
optymalnego
przesunięcia
w algorytmie KMP



Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz wzorec 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 wzorec, 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.

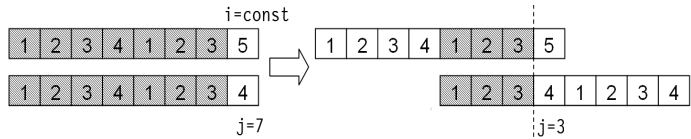
Zanim zagłębimy się w wyjaśnienia na temat obliczania tych przesunięć, popatrzmy na efekt ich działania na kilku kolejnych przykładach. Na rysunku 14.4 możemy

³ Lub i w przypadku badanego tekstu.

dostrzec, że na siódmej pozycji wzorca⁴ (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

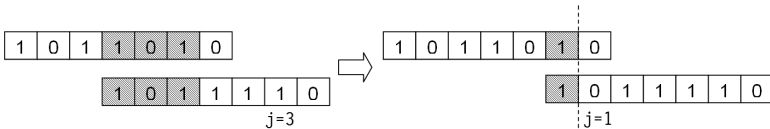
RYСУNEK 14.4.

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.



RYСУNEK 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 niezmienny, 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!

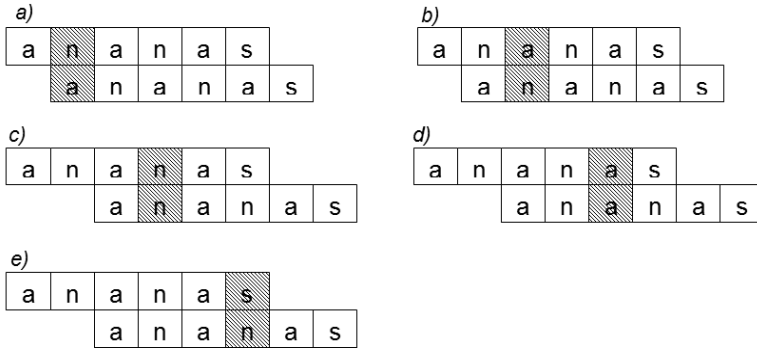
```

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).



RYСУNEK 14.6. Optymalne przesunięcia wzorca „anas” 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.

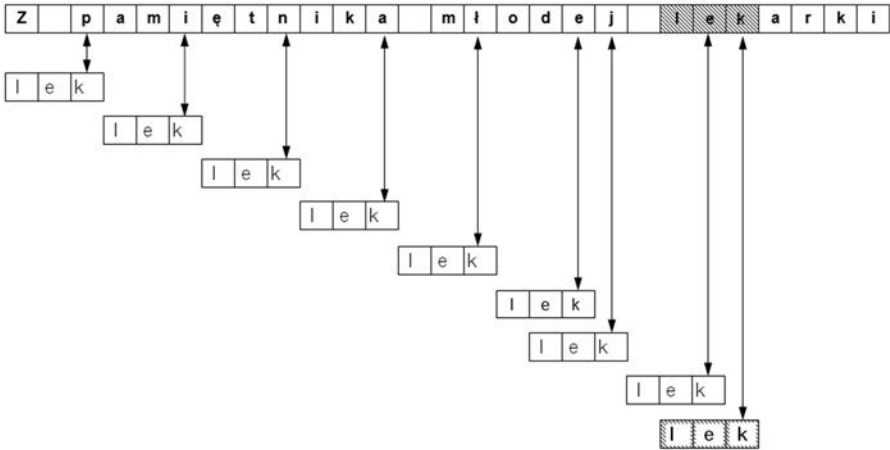
Użycie tego algorytmu jest jednak niezbędne w tych aplikacjach, w których następuje liniowe przeglądanie tekstu — bez buforowania. Jak łatwo zauważyć, wskaźnik `i` w funkcji `kmp` nigdy nie jest dekrementowany, co oznacza, że plik można przeglądać od początku do końca bez cofania się w nim. W niektórych 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.



RYСУNEK 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 przesunął wzorec 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 zbyt 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"
# 012345678901234567890123456789
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 przesunięć⁶ 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:

- Wzorec 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 znaków⁷. 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.

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

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

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

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

Jeśli dobrze przyjrzy 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. Założmy, 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:

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

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): # Wyliczmy 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 sfalszowałaby końcowy wynik!

Kolejne uwagi dotyczą parametrów `p` i `MAX`. Zaleca się, aby `p` było dużą liczbą pierwszą⁸, 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 $p \ll 6$.

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 założmy, że zdania podajemy bez dużych liter.

ZADANIE 3

Napisz program sprawdzający, czy zdanie wejściowe zawiera ukryte słowo. Przykładowo, tekst „Broniek alergicznie nie znosił makaronu z kaszą” 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):
    d1 = len(s)
    cpt = 0
    test = True # 's' jest (na razie) palindromem
    while cpt <= d1 // 2 and test == True:
        if s[cpt] == s[d1-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', 'l', 'o', 'n', ' ', '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', 'l', 'o', 'n', ' ', '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

(1,), 151
** (operator), 40
.shape, 242
// (operator), 40
@property, 163,
 Patrz właściwości
add, 167
add(), 187
eq(), 305
hash(), 305
init, 158, 162
init.py, 191
mul, 167
pow, 167
sub, 167
truediv, 167
1D, 236
2D, 236
3DES, 462

A

A*, 502
A* (strategia), 426
Ackermann.py, 127
acykliczny graf
 skierowany, 398
add(), 225, 241, 436
Adleman L., 463
Aiken A., 29
alfabet Braille'a, 473
alfabet Morse'a, 473, 474
algebra Boole'a, 55
Algorismus, *Patrz*
 Muhammad ibn Musa
 al-Chuwarizmi
algorytm
 A*, 502
 Bellmana-Forda, 418
 cechy, 25
 cięcie α - β , 502, 503

DES, 462
deterministyczny, 26
Dijkstry, 416
Euklidesa, 34
Fleury'ego, 400
Floyda, 414
Floyda-Warshalla,
 413
genetyczny, 392
heurystyczny, 390
 operatory, 390
 stany, 390
Huffmana, 477, 480
Kruskala, 419
kryteria wyboru, 105
LZW, 483, 484
mini-max, 502, 511
niestabilny, 98
poziomy abstrakcji,
 33
 prefiksowy, 477
Prima, 419, 420
Rabina i Karpa, 359
Quicksort, 318
SSS*, 502
Strassena, 372
Warshalla, 409
algorytmy numeryczne,
 433
 całkowanie funkcji
 metodą Simpsona,
 443
 interpolacja funkcji
 metodą Lagrange'a,
 439
 iteracyjne obliczanie
 wartości funkcji, 438
 metoda Gaussa, 444
 metoda Stirlinga, 441
 poszukiwanie miejsc
 zerowych funkcji,
 437

rozwiązywanie
 układów równań
 liniowych, 444
 różniczkowanie
 funkcji, 441
algorytmy
 przeszukiwania
 hashing, 291
 przeszukiwanie
 liniowe, 287
 transformacja
 kluczowa, 291
 algorytmy sortowania,
 311
 duże pliki wejściowe,
 323
 klasa $O(N^2)$, 312, 314
 kryteria wyboru
 algorytmu, 328
 Quicksort, 316
 Shaker-sort, 315
 sortowanie
 bąbelkowe, 314
 sortowanie przez
 scalanie, 321
 sortowanie przez
 wstawianie, 312
 sortowanie przez
 wytrząsanie, 315
 sortowanie
 zewnętrzne, 309,
 322, 326
 algorytmy żarłocze,
 374
 problem plecakowy,
 375
 schemat algorytmu,
 375
Anagramy.py, 363
Anakonda, 513
analiza

bezpieczeństwa sieci,
394
obwodów
elektronicznych,
394
wyrażenia
beznawiasowego,
275
analiza złożoności
algorytmów, 105
analizaWWW.py, 227
analizaWWW2.py, 231
append(), 222
arange(), 238
arytmetyka dużych
liczb, 458, 467
ASCII, 65, 349, 457, 477
asin(), 435
atan(), 435
automat skończony, 397
automatyczne
testowanie, 30

B

Babbage C., 27
Base64, 466
BASIC, 338
baza wiedzy, 493, 495
BazaDanych.py, 195,
196
BazaDanychMain.py,
198, 200
Bellman R. E., 380, 418
BFS, 421
big endian, 73
bin(), 57
binarne drzewo
poszukiwań, 268
binarnie.py, 59
BinarySearchIter.py,
290
BinSearch.py, 97
BinSearch2.py, 124
BM.py, 357
BMP, 69
bool, 137
Boyer R. S., 352

BPMN, *Patrz* Business
Process Modeling
Notation
Brackets, 520
Braille L., 474
breadth-first search, 421
break, 42
Bronek.py, 363
brute-force, 472
BST, *Patrz* binarne
drzewo poszukiwań
BST.py, 268
BSTMain.py, 272
BULL GAMMA 3, 29
Business Process
Modeling Notation, 39

C

cache, 130
całkowanie funkcji, 443
metoda Simpsona,
443
Carnota L., 27
ceil(), 435
char, 64
ciąg Fibonacciego, 81,
133, 381
programowanie
dynamiczne, 381
ciągi znaków,
reprezentacja, 64
ciągła integracja, 30
cięcie α - β , 502, 503
clear(), 222, 225, 232
COBOL, 329
color, 450
complex, 137, 138
complex0.py, 157, 158
complex128, 239
complex64, 239
continue, 42
Cook S. A., 351
COPACOBANA, 463
cos(), 435
count(), 151, 222
CP-1250, 67
CSV, 234

cykl
Eulera, 399, 400
Hamiltona, 398
skierowany, 398
czas jednostkowy
wykonania instrukcji,
118
czas wykonania, 106,
111
częstkowy dobór, 428

D

DAG, 398, *Patrz*
acykliczny graf
skierowany
DARPA, 493
Data Encryption
Standard, 462
debugger, 521
def, *Patrz* funkcja
degress(), 435
dekompozycja
problemu, 90, 369
dekorator, 163, 181
del, 231
depozyt.py, 161, 164
depth-first search, 421
derekursywacja, 329,
333
eliminacja
zmiennych
lokalnych, 338
metoda funkcji
przeciwnych, 340
schemat, 342
typu if-else, 344
typu while, 343
z podwójnym
wywołaniem
rekurencyjnym,
346
stos, 337
DES, 462
DFS, 421
DIB, 69
difference(), 225
Diffie W., 463

- digraf, 396
 - Dijkstra E., 32, 416
 - directed acyclic graph, 398
 - directed graph, 396
 - divide(), 241, 436
 - DNA, 385
 - do... while (pętla), 44
 - dobor.py, 429
 - Doctor Who, 229
 - dokumentacja Pythona, 531
 - domknięcie
 - przechodnie grafu, 408
 - drzewa, 263
 - gry, 500, 501
 - kodowe, 478
 - korzeń, 264
 - liść, 264
 - m-drzewa, 264
 - ojciec, 265
 - potomek lewy, 265
 - potomek prawy, 265
 - realizacja tablicowa, 267
 - reprezentacja
 - w Pythonie, 266
 - rozpinające
 - minimalne, 419
 - syn, 265
 - Uniwersalna
 - Struktura
 - Słownikowa, 279
 - uporządkowane, 264
 - węzły, 264
 - końcowe, 264
 - wysokość, 264
 - drzewa binarne, 174, 255, 266, 273
 - komórka, 266
 - wypisywanie drzewa
 - w postaci
 - wrostkowej, 277
 - wrażenia
 - arytmetyczne, 273
 - wysokość, 265
 - dtype, 238
 - dyn2D.py, 383
 - dziedzic.py, 169
 - dziedziczenie klas, 168
 - dziel i zwyciężaj, 321, 322, 366
 - algorytm Strassena, 372
 - mnożenie liczb
 - całkowitych, 373
 - mnożenie macierzy, 370
 - odnajdywanie
 - największego
 - i najmniejszego
 - elementu w tablicy, 367
 - Quicksort, 374
 - dzielenie całkowite, 40
- ## E
- Eckert J. P., 29
 - EDVAC, 29
 - edytory do Pythona, 519
 - eliminacja Gaussa, 444
 - eliminacja zmiennych
 - lokalnych, 339, 341
 - end-recursion, 332
 - ENIAC, 29
 - Eratostenes.py, 52
 - Euklides, 23
 - Euler L., 393
 - except, 163
 - exp(), 435
 - extend(), 222
- ## F
- fabs(), 435
 - fib.py, 82
 - fibdyn.py, 382
 - FIFO, 251, 253
 - FIFO.py, 252
 - First In First Out, 251
 - Flawiusz Józef
 - (zagadka), 209
 - Fleury H., 400
 - float, 64, 137
 - float(), 142
 - float16, 239
 - float32, 239
 - float64, 239
 - floor(), 297, 435
 - Floyd R., 32
 - for, 42, 75
 - Ford L. R. Jr., 418
 - format GIF, 486
 - formatowanie liczb, 146, 147
 - formatowanie.py, 144
 - Forth, 276
 - Fortran, 329, 447
 - fraktale, 75
 - funkcja, 41
 - Ackermanna, 127
 - McCarthy'ego, 84, 98
 - O, 113, 114
 - odwrotna, 340
 - funkcje
 - H, 293, 294, 360
 - mnożenie, 296
 - suma modulo 2, 295
 - suma modulo R_{max} , 296
 - matematyczne, 434, 436
 - modulo, 360
 - wywołanie przez
 - wartość, 87
 - zwracanie wielu
 - wartości, 153
 - fuzja list, 188
- ## G
- Garmisch, 31
 - Gauss.py, 445
 - GCD, 100
 - get(), 232
 - GIF, 69, 458, 483, 486
 - GIF87a, 486
 - GIG89a, 486
 - GitHub, 533
 - głowa listy, 176

GNU Scientific Library, 434
 Go, 502
 Gödel K., 28
 Goldman A., 400
 Gosper R. W., 352
 goto, 330
 gra w „kółko i krzyżyk”, 502
 grafika rastrowa, 69
 grafika wektorowa, 69
 grafy, 174, 393
 acykliczne
 skierowane, 398
 algorytm
 Bellmana-Forda, 418
 Dijkstry, 416
 Fleury’ego, 400
 Floyda, 414
 Floyda-Warshalla, 413
 Kruskala, 419
 Prima, 419, 420
 Roy-Warshalla, 409
 cykl, 398
 Eulera, 399, 400
 Hamiltona, 398
 DAG, 398
 diagonalne, 408
 digraf, 396
 domknięcie
 przechodnie, 408
 eulerowskie, 399
 implementacja, 402
 kompozycja, 406
 liczba chromatyczna grafu, 396
 macierz sąsiedztwa, 401
 minimalizowanie konfliktów, 426
 minimalne drzewo rozpinające, 419
 nieskierowane, 397
 operacje, 406
 matematyczne, 406
 parzysty stopień, 400

planarne, 395
 poszukiwanie drogi, 411
 problem doboru, 426
 problem
 komiwojażera, 399
 przechodnie, 408
 przeszukiwanie, 421
 w głąb, 421, 422
 wszerz, 421, 424
 zstępujące, 422
 regularne, 396
 reprezentacja, 401
 tablicowa, 401
 skierowane, 396
 słownik węzłów, 401, 404
 spójne, 397
 stanów, 500
 stopień wejściowy wierzchołka, 396
 suma, 406
 symetria, 408
 tablica
 dwuwymiarowa, 401
 węzeł, 395
 początkowy, 395
 wierzchołki, 395
 zbiory, 405
 greedy, *Patrz*
 algorytmy żarłoczne
 greedy.py, 377
 GSL, *Patrz* GNU Scientific Library

H

H (funkcja), 293, 360
 hano3.py, 347
 Hanoi, 333, 339
 hanoi.py, 335
 hanoi2.py, 336
 hashcat, 473
 HashHashBaby.py, 307
 hashing, 291
 heap, *Patrz* sterta
 Hellman M., 463

hermetyzacja, 156, 160
 heurystyka, 389, 421
 hex(), 57
 Hilberta D., 28
 hill-climbing, 426
 Hoare T., 32
 Hollerith H., 28
 hstack(), 244
 Huffman(), 482

I

IBM, 28
 IBM 604, 29
 IBM 650, 29
 ICO, 69
 IDLE, 521
 IFIP, 31
 implementacja grafów, 402
 in, 221, 231
 index(), 151, 222
 input(), 128, 507
 insert(), 222
 int, 64, 137
 int(), 142
 int16, 239
 int32, 239
 int64, 239
 interpolacja funkcji, 439
 metodą Lagrange’a, 439
 interpolacja.py, 440
 intersection(), 225
 ISO 8859-2, 67
 ISO Latin-2, 67
 issubset(), 226
 issuperset(), 226
 iteracja, 42, 75
 iteracyjne obliczanie wartości funkcji, 438
 iterator, 210

J

Jacquard J. M., 27
 JPEG, 69

K

Karp R. M., 352
 Kartoteka.py, 195
 keys(), 232
 klasa, 155
 algorytmu, 109
 bazowa, 168
 O, 114
 projektowanie, 160
 klucz, 292, 359, 405
 prywatny, 463
 publiczny, 463
 K-M-P (algorytm), 352
 tablica przesunięć, 354
 KMP.py, 354
 Knuth D. E., 352
 kod
 ASCII, 477
 znaki
 podstawowe, 66, 67
 Cezara, 458
 Huffmana, 457, 477, 479
 nadmiarowy, 457
 nierównomierny, 457
 równomierny, 457
 Znak-Moduł, 62
 kodCezara.py, 458
 kodowanie danych, 455, 458
 3DES, 462
 asymetryczne, 463
 DES, 462
 klucz prywatny, 463
 klucz publiczny, 463
 LZW, 458, 483
 podpis cyfrowy, 464
 problem transmisji klucza, 463
 RSA, 464
 symetryczne, 461
 z kluczem publicznym, 457, 463
 za pomocą słownika, 483

kodowanie
 liter za pomocą 5 bitów, 294
 obrazów, 68
 znaków, 65
 kody ASCII, 457
 KodyProste.py, 460
 kolejka, 424
 FIFO, 251
 LIFO, 247
 priorytetowa, 254
 operacje, 254
 KolejkaMain.py, 253
 kolejności działań, 40
 kompilator, 330, 394
 kompletne drzewo binarne, 255
 kompozycja grafów, 406
 kompresja, 455, 456, 473
 algorytm Huffmana, 480
 algorytmy, 474
 bezstratna, 474
 danych, 280, 281
 GIF, 458, 483, 486
 LZW, 483
 metoda Huffmana, 477
 poprzez modelowanie matematyczne, 475
 redundancja, 475
 RLE, 476
 stopień kompresji, 475
 stratna, 474
 szybkość działania, 475
 komputer kwantowy, 471
 konstruktor, 158
 kontroler wyводу, 495
 konwersje napisów na liczby, 142
 korzeń (drzewa), 264

kółko i krzyżyk, 501, 502, 504
 generowanie listy możliwych ruchów, 509
 kodowanie listy węzłów potomnych, 509
 linie otwarte, 504
 Kruskal J., 419
 kryptografia, 455
 kryptosystem RSA, 463
 kwadraty „parzyste”, 94
 kwadraty.py, 94

L

Last In First Out, 247
 LCS, 385
 LCS.py, 386
 LCS2.py, 388
 Lempel A., 483
 len(), 221, 231
 len(x), 221
 LF, 457
 liczby, reprezentacja, 64
 LIFO, 247
 linestyle, 450
 linewidth, 450
 linspace(), 241
 Lisp, 189, 495
 list comprehension, 223
 list(), 219
 lista
 implementacja tablicowa, 200
 jednokierunkowa, 192
 wady i zalety, 192
 sąsiedztwa, 401
 Lista.py, 191
 lista1.py, 177
 lista2.py, 179, 181, 182
 lista2iter.py, 211
 Lista2Kier.py, 206
 Lista2KierMain.py, 208

ListaMain.py, 191
 ListaTabMain.py, 202
 list-tab.py, 201, 202
 listy, 174
 cykliczne, 209
 dwukierunkowe, 206
 jednokierunkowe,
 175
 głowa, 176, 178
 ogon, 176
 rankingowe, 427
 listy2.py, 187
 liść (drzewa), 264
 litte endian, 73
 log(), 435
 Logo, 33
 longest common
 subsequence, 385
 Lovelace A., 28
 Lucas É., 333
 LZW, 458, 483
 kodowanie, 484

Ł

łamanie szyfrów, 471
 brute-force, 472
 dedukcja na
 podstawie
 fragmentu lub
 całości treści, 472
 kradzież klucza, 472
 metoda „na siłę”, 472

M

macierz
 kierowania ruchem,
 410
 sąsiedztwa, 401
 trójkątna, 444
 macOS, 515
 maksimum w tablicy
 liczb, 367
 MARK 1, 29
 marker, 450
 Markow A., 28

maszyna
 Moore'a, 397
 Turinga, 35
 Mathcad, 433
 Mathematica, 433
 Matlab, 433
 matplotlib, 433
 Matplotlib, 434, 447
 Mauchly J. W., 29
 max(), 240
 McCarthy J., 31, 91,
 Patrz funkcja
 McCarthy'ego
 McCarthy.py, 84
 m-drzewa, 264
 mean(), 240
 MergeSort.py, 321
 metadane, 70
 metoda, 155
 funkcji przeciwnych,
 340
 Gaussa, 444
 Huffmana, 477
 Newtona, 437
 Simpsona, 443
 Stirlinga, 441
 transformacji
 kluczowej, 308
 metodologia
 programowania, 365
 metody prymitywne,
 458
 miara złożoności
 obliczeniowej, 107
 Microsoft Visual C++
 Redistributable, 448
 Microsoft Visual Studio,
 530
 miejsca zerowe funkcji,
 437
 min(), 240
 minimalizowanie
 konfliktów, 426
 minimalne drzewo
 rozpinające, 419
 mini-max, 502, 511
 minimum w tablicy
 liczb, 367

min-max1.py, 367
 min-max2.py, 368
 Mitnick, 472
 mnożenie liczb
 całkowitych, 373
 mnożenie macierzy, 370
 modelowanie
 matematyczne, 475
 modelowanie
 rozwiązywanego
 zagadnienia, 497
 modulo, 40, 360, 470
 moduł dialogowy, 495
 Moore J. S., 352
 Morris J. H., 352
 Morse S., 473
 motor wywodu, 495
 Muhammad ibn Musa
 al-Chuwarizmi, 23
 multiply(), 241, 436
 MYCIN, 494
 myślenie rekurencyjne,
 90

N

n, *Patrz* znak nowej linii
 n.py, 87
 najbardziej czasochłonna
 operacja, 110, 118
 najdłuższa wspólna
 podsekwencja, 385
 napisy.py, 142
 ndarray, 235
 negative(), 241, 436
 Newton.py, 438
 normalizowanie osi
 wykresu, 450
 notacja dużego O, 114
 notacja Landaua, 114
 NumPy, 234, 434
 numpy0.py, 237
 numpy1.py, 241, 243
 numpy3.py, 245
 NWD, 34, 100
 nwd.py, 34, 100
 nwd2.py, 103

O

$O(1)$, 113
 $O(2^n)$, 114
 $O(\log n)$, 113
 $O(n)$, 113
 $O(n^2)$, 113, 114
 $O(n^3)$, 113
 obiekt, 155
 Object Oriented Programming, 155
 obliczanie wartości funkcji, 438
 obwód zamknięty, 409
 odczytBMP.py, 73
 odległość Levenshteina, 385
 odnajdywanie największego i najmniejszego elementu w tablicy, 367
 OdwrocTab.py, 101
 Odwrotna Notacja Polska, 275, 277
 odwrotna.py, 341
 odwrotna2.py, 346
 ogon listy, 176
 ojciec, 265
 O-notacja, 113, 114, 115
 ONP, *Patrz* Odwrotna Notacja Polska
 OOP, *Patrz* Object Oriented Programming
 operacje na grafach, 406
 operandy, 274
 operatory, 40, 274
 arytmetyczne
 modelowanie w klasie, 166
 dostępu, 221
 optymalizacja programów, 130

P

PageRank, 394
 Palindromy.py, 362
 pamięć komputerowa, 311
 Pandas, 434
 Pascal B., 27
 petenci.csv, 305
 petle.py, 43
 pętla, 75
 nieskończona, 42
 pi (stała), 435
 pierwsza.py, 52
 pip, 217, 447, 518
 instalacja pakietów w PyCharm, 529
 pliki GIF, 488
 plot(), 235, 450
 PNG, 69
 pochodne.py, 442
 podejście
 iteracyjne, 75
 zorientowane obiektowo, 155
 podejżdżanie (metoda), 426
 podpis cyfrowy, 464
 podwójne kluczowanie, 301, 302
 pola, 155
 pomiarczasu.py, 131
 pop(), 222, 225, 232, 248, 249
 porównywanie sekwencji kodów, 385
 PostacDwojkowa.py, 102
 postać uwikłana, 438
 Postscript, 276
 poszukiwanie miejsc zerowych funkcji, 437
 potegi.py, 104
 potęga grafu, 407
 potęgowanie, *Patrz* ** (operator)
 potomek lewy, 265

potomek prawy, 265
 pow(), 435
 power(), 241, 436
 Pratt V. R., 352
 prawdopodobieństwa występowania liter w języku polskim, 480, 485, 486
 prefiks, 478
 Prim R. C., 420
 private, 172, 249
 problem
 8 hetmanów, 391
 doboru, 426, 427, 429
 cząstkowy dobór, 428
 listy rankingowe, 427
 komiwojażera, 399
 konika szachowego, 498
 NP-zupełny, 399
 optymalnego doboru, 394
 plecakowy, 375
 transmisji klucza, 463
 wyrażania preferencji, 427
 proces koncepcji programów, 32
 program rekurencyjny, 78, 88, 121
 programowanie dynamiczne, 83, 379
 ciąg Fibonacciego, 381
 etapy, 380
 najdłuższa wspólna podsekwencja, 385
 równania z wieloma zmiennymi, 383
 Prolog, 495
 prostota algorytmu, 105
 protected, 172, 249
 próbkowanie liniowe, 299, 308

przedrostek, 478
 przepełnienie stosu, 83, 128
 przeszukiwanie, 287
 binarne, 96, 124, 290, 374
 klasa algorytmu, 125
 grafów, 421
 A*, 426
 BFS, 421
 DFS, 421
 ekspansja węzłów, 422
 metoda
 podjeżdżania, 426
 strategie, 426
 w głąb, 421, 422
 wszerz, 421, 424
 z powracaniem, 426
 zstępujące
 przeszukiwanie, 422
 liniowe, 287
 tekstów
 algorytm K-M-P, 352
 algorytm typu
 brute-force, 349
 przypadek
 najgorszy, 119
 najlepszy, 119
 średni, 120
 typowy, 120
 public, 172
 push(), 247, 249, 250
 Pycharm, 448, 521
 PyCharm, 525
 instalacja pakietów, 530
 Python, instalacja, 513

Q

Quicksort, 316, 374
 podział tablicy, 317

R

Rabin M. O., 352
 radians(), 435
 RAM, *Patrz* pamięć komputerowa
 range(), 44
 ravel(), 243
 RC, *Patrz* równanie charakterystyczne
 RecursionError, 128
 redukcja wsteczna, 444, 445
 redundancja, 475
 referencje, 139
 do funkcji, 314
 referencje.py, 139
 reguła mini-max, 503
 rekurencja, 75, 121, 365
 ciąg Fibonacciego, 81
 dekompozycja problemu, 90
 drzewo wywołań, 80
 ilustracja, 77
 kontekst, 89
 kwadraty „parzyste”, 94
 myślenie rekurencyjne, 90
 naturalna, 88
 nieskończona ilość wywołań, 85
 pamięciożerność, 98
 poprawność definicji, 87
 poziomy, 79, 80, 89
 problemy, 80
 przebiegi, 75
 przekazywanie parametrów, 80
 przepełnienie stosu, 83
 rozkład na problemy elementarne, 76
 silnia, 88
 skrośna, 98
 spirala, 90
 sposób wykonywania programu, 78
 typy programów, 88
 uwagi praktyczne, 98
 z parametrem dodatkowym, 88, 89, 336
 zajętość pamięci, 83
 zakończenie algorytmu, 76
 zakończenie programu, 78
 zasada działania, 76
 zmniejszanie rozmiaru problemu, 98
 rekursja, *Patrz* rekurencja
 relacje binarne, 408
 relaksacja, 417
 remove(), 222, 225
 reprezentacja grafów, 401
 reprezentacja tablicowa grafów, 422
 reshape(), 241, 242
 resize(), 243
 reszta.py, 379
 reverse(), 222
 reverse-engineeringu, 462
 Reversi, 502
 RGB, 450
 Ritchie D., 30
 Rivest R., 463
 RK.py, 361
 RLE, 476
 RO, *Patrz* rozwiązanie ogólne
 routing matrix, 410
 rozdzielenie i scalanie połączone z sortowaniem, 323
 rozkład logarytmiczny, 124
 rozmiar danych, 108
 rozwiązanie ogólne, 122

rozwiązanie
 równania
 rekurencyjnego, 123
 szczególne, 122
 rozwiązywanie układów
 równań, 444
 liniowych, 444
 równanie
 charakterystyczne, 122
 z wieloma
 zmiennymi, 383
 różniczkowanie funkcji, 441
 RS, *Patrz* rozwiązanie
 szczególne
 RSA, 462, 463
 ruchy dozwolone, 498
 Run Length Encoding, 476

S

scalanie zbiorów
 posortowanych, 320
 Scalanie.py, 320
 schemat blokowy, 38
 blok decyzyjny, 39
 blok obliczeniowy, 39
 pętle, 38
 schemat Hornera, 438, 468
 schematy
 derekursywacji, 342
 if-else, 344
 while, 343
 z podwójnym
 wywołaniem
 rekurencyjnym, 346
 SciPy, 434
 sdw.py, 85
 Segmentation fault, 86
 self, 158
 set comprehension, 228
 setter, 163

Shaker-sort, 315
 Shamir A., 463
 SI, *Patrz* Sztuczna
 Inteligencja
 sieci neuronowe, 495
 nauczanie, 497
 struktura, 496
 silnia (definicja), 79
 silnia, 79, 88, 108, 110,
 111, 123, 336
 silnia.py, 79
 silnia.py, 88
 silnia2.py, 89, 336
 silnia-it.py, 104
 silnia-it.py, 337
 Simpson.py, 443
 sin(x), 435
 sito Erastotenesa, 49
 słowniki.py, 233
 słowniki, 228, 279
 węzłów, 404, 422
 słowniki0.py, 229
 sort(), 222, 240
 sorted(), 327
 sortMain.py, 314
 sortowanie
 algorytm klasy
 $O(N^2)$, 312, 314
 algorytmy, 311
 bąbelkowe, 314
 przez scalanie, 321
 przez wstawianie,
 312
 przez wytrząsanie,
 315
 sterta, 262
 szybkie, 316
 wewnętrzne, 311
 zewewnętrzne, 309,
 311, 322, 326
 sortPython, 326
 spirala, 90
 spirala.py, 92
 sqrt(), 241, 435, 436
 SRL, *Patrz* szereg
 rekurencyjny liniowy
 SSS*, *Patrz* algorytm
 SSS*

Stack overflow, 83, 86
 stałe symboliczne, 138
 sterta, 254, 255
 implementacja, 258
 kolejka priorytetowa,
 257
 sortowanie, 262
 tworzenie, 256
 wstawianie
 elementów, 257,
 259
 Sterta.py, 258
 StertaMain.py, 261
 StertaSort.py, 262
 stopień kompresji, 475
 stopień wejściowy
 wierzchołka grafu, 396
 stos, 247
 pop(), 248
 push(), 247, 250
 zasada działania,
 247, 248
 StosMain.py, 250
 StosOgraniczony.py,
 249
 str(), 143
 Stranger Things, 338
 Strassen V., 371
 strategia gry, 501
 strategia
 przeszukiwania, 502
 struktury danych, 174
 drzewa, 263
 grafy, 393
 kolejka FIFO, 251
 kolejka priorytetowa,
 254
 listy
 jednokierunkowe,
 175
 sterta, 254
 stos, 247
 zbiory, 213
 Sublime Text, 520
 subtract(), 241, 436
 sum(), 240
 suma grafów, 406
 suma modulo 2, 295

suma moduło R_{\max} , 296
 SVG, 69
 symetria.py, 332
 symmetric_difference(), 226
 symulacja inteligentnego zachowania, 491
 syn, 265
 system
 dziesiąty, 54
 ekspercki, 493
 baza wiedzy, 493
 kontroler wyvodu, 495
 moduł dialogowy, 495
 motor wyvodu, 495
 MYCIN, 494
 sztuczny lekarz, 494
 kontrolni wersji, 30
 obliczeniowy, 26
 ósemkowy, 57
 szesnastkowy, 56
 uzupełnienia dwójkowego, 63
 systemsort.py, 324
 szachy, 502
 szereg rekurencyjny liniowy, 122
 Sztuczna Inteligencja, 491, 492
 cele, 492
 drzewa gier, 500
 grafy stanów, 500
 sieci neuronowe, 495
 sztuczny lekarz, 494
 szukaj.py, 118
 SzukajLiniowo.py, 287
 szukaj-rek.py, 77
 szukajTXT.py, 350
 szukamy.py, 306
 Szumańska E., 357
 szyfr blokowy, 462

T

t, *Patrz* tabulator
 TabInt.py, 313, 314
 tablica dwuwymiarowa, 401
 tablica przesunięć, 354
 tablica różnic centralnych, 441
 tablice równoległe, 267
 tablice.py, 148
 tablice2.py, 149
 tabulator, 142
 tan(x), 435
 techniki optymalizacji programów, 130
 techniki programowania, 365
 teoria automatów, 397
 teoria gier, 421
 test Turinga, 492
 Thompson K., 30
 TicTacToe.py, 505, 509
 time(), 131
 transformacja kluczowa, 291, 308
 funkcje H, 293
 podwójne kluczowanie, 301
 próbkiwanie liniowe, 299
 strefa podstawowa, 299
 strefa przepełnienia, 299
 tablice, 298
 zastosowania, 303
 transpose(), 244
 traveling salesman problem, 399
 trunc(), 435
 try, 163
 tupla 150, 185
 modyfikacja, 151
 tuple.py, 151
 Turing A., 28, 35, 492

typy danych, 135
 proste, 135
 wbudowane typy złożone, 136
 typy złożoności obliczeniowej, 118

U

U2, *Patrz* system uzupełnienia dwójkowego
 uint16, 239
 uint32, 239
 uint64, 239
 uint8, 239
 UML, *Patrz* Unified Modeling Language
 unfunc, 240, 436
 Unicode, 68, 349
 UTF-16, 68
 Unified Modeling Language, 39
 union(), 225
 UNIVAC 1, 29
 Uniwersalna Struktura Słownikowa, 279
 UNIX, 30
 UNIX (czas narodzin), 131
 update(), 232
 USS, *Patrz* Uniwersalna Struktura Słownikowa
 USS.py, 280
 USSMain.py, 284
 UTF-16, *Patrz* Unicode

V

ValueError, 128, 507
 values(), 232
 Vim, 520
 vstack(), 244

W

WartFun.py, 439
 Welch T., 483
 while, 42, 75
 wielomiany, 468
 wielomiany.py, 468
 wielowątkowość, 130
 wieże Hanoi, 333, 339,
 347, 374
 Wirth N., 32
 właściwości, 163
 wskaźnikiFunkcji.py,
 288
 wycinki, 244
 wydajność
 oprogramowania, 107
 wykres, formatowanie,
 449
 wykres1.py, 451
 wykres3.py, 453
 wykresy, 448
 wypiszbity.py, 60
 WyrażeniaMain.py, 275
 wyrażanie preferencji,
 427

wyrażenia
 arytmetyczne, 273,
 275
 wysokość drzewa
 binarnego, 265
 wywołanie przez
 wartość, 87
 wywołanie terminalne,
 332
 wyznacznik
 Vandermonde'a, 440
 wzór Simpsona, 443
 wzór Stirlinga, 441

X

XOR, 460

Z

zajętość pamięci, 106
 zamiana dziedziny
 równania
 rekurencyjnego, 127
 zapis szesnastkowy, 56
 zasięgi.py, 140

zasięg zmiennych, 140
 ZbiorLitery.py, 214
 zbiory, 213, 223, 401
 zbiory.py, 223
 zerowanie fragmentu
 tablicy, 116
 Ziv J., 483
 złożoność
 algorytmów, 105
 czasowa, 107
 obliczeniowa, 105,
 125
 pamięciowa, 107,
 132
 praktyczna, 111
 teoretyczna, 112
 zmienne, 64, 136
 globalne, 338
 klasowe, 162
 lokalne, 338
 zasięg, 140
 zmienne.py, 137
 znak nowej linii, 142
 znaki, 65

PROGRAM PARTNERSKI

— GRUPY HELION —

- 
1. ZAREJESTRUJ SIĘ
 2. PREZENTUJ KSIĄŻKI
 3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion

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 Brzoński-Brzońskiewicza, gdańskiego performerera 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

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-9368-4



9 788328 393684

Cena: 119,00 zł