

algorytmy

Ilustrowany przewodnik

Aditya Y. Bhargava



Helion



Tytuł oryginału: Grokking Algorithms: An illustrated guide
for programmers and other curious people

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-3446-5

Original edition copyright © 2016 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2017 by HELION SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopowanie 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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/algoip.zip>

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

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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







Dla moich rodziców, Sangeety i Yogesha





Spis treści

Przedmowa	xiii
Podziękowania	xiv
O książce	xv
1. Wprowadzenie do algorytmów	1
Wprowadzenie	1
Czego nauczysz się o wydajności	2
Czego nauczysz się o rozwiązywaniu problemów	2
Wyszukiwanie binarne	3
Lepszy sposób wyszukiwania	5
Czas wykonywania	10
Notacja dużego O	10
Czas wykonywania algorytmów	
rośnie w różnym tempie	11
Wizualizacja różnych czasów wykonywania	13
Notacja dużego O określa	
czas działania w najgorszym przypadku	15
Kilka typowych czasów wykonywania	15
Problem komiwojażera	17
Powtórzenie	19
2. Sortowanie przez wybieranie	21
Jak działa pamięć	22
Tablice i listy powiązane	24
Listy powiązane	25
Tablice	26

Terminologia	27
Wstawianie elementów w środku listy	29
Usuwanie elementów	30
Sortowanie przez wybieranie	32
Powtórzenie	36
3. Rekurencja	37
Rekurencja	38
Przypadki podstawowy i rekurencyjny	40
Stos	42
Stos wywołań	43
Stos wywołań z rekurencją	45
Powtórzenie	50
4. Szybkie sortowanie	51
„Dziel i rządź”	52
Sortowanie szybkie	60
Jeszcze raz o notacji dużego O	66
Sortowanie przez scalanie a sortowanie szybkie	67
Przypadki średni i najgorszy	68
Powtórzenie	72
5. Tablice skrótów	73
Funkcje obliczania skrótów	76
Zastosowania tablic skrótów	79
Przeszukiwanie tablic skrótów	80
Zapobieganie powstawaniu duplikatów elementów	81
Tablice skrótów jako pamięć podręczna	83
Powtórzenie wiadomości	86
Kolizje	86

Wydajność	88
Współczynnik zapełnienia	90
Dobra funkcja obliczania skrótów	92
Powtórzenie	94
 6. Przeszukiwanie wszerz	95
 Wprowadzenie do grafów	96
Czym jest graf	98
Wyszukiwanie wszerz	99
Szukanie najkrótszej drogi	102
Kolejki	103
Implementacja grafu	105
Implementacja algorytmu	107
Czas wykonywania	111
Powtórzenie	114
 7. Algorytm Dijkstry	115
Posługiwanie się algorytmem Dijkstry	116
Terminologia	120
Szukanie funduszy na fortepian	122
Krawędzie o wadze ujemnej	128
Implementacja	131
Powtórzenie	140
 8. Algorytmy zachłanne	141
Plan zajęć w sali lekcyjnej	142
Problem plecaka	144
Problem pokrycia zbioru	146
Algorytmy aproksymacyjne	147
Problemy NP-zupełne	152
Problem komiwojażera krok po kroku	153

Trzy miasta	154
Cztery miasta	155
Jak rozpoznać, czy problem jest NP-zupełny	158
Powtórzenie wiadomości	160
9. Programowanie dynamiczne	161
Problem plecaka	161
Proste rozwiązańe	162
Programowanie dynamiczne	163
Pytania dotyczące problemu plecaka	171
Co się dzieje, gdy zostanie dodany element	171
Jaki będzie skutek zmiany kolejności wierszy	174
Czy siatkę można wypełniać wg kolumn zamiast wierszy	174
Co się stanie, gdy doda się mniejszy element	174
Czy można ukraść ułamek przedmiotu	175
Optymalizacja planu podróży	175
Postępowanie z wzajemnie zależnymi przedmiotami	177
Czy możliwe jest, aby rozwiązanie wymagało więcej niż dwóch podplecaków	177
Czy najlepsze rozwiązanie zawsze oznacza całkowite zapełnienie plecaka?	178
Najdłuższa wspólna część łańcucha	178
Przygotowanie siatki	179
Wypełnianie siatki	180
Najdłuższa wspólna podsekwencja	183
Najdłuższa wspólna podsekwencja — rozwiązanie	184
Powtórzenie	186
10. K najbliższych sąsiadów	187
Klasyfikacja pomarańczy i grejpfrutów	187
Budowa systemu rekommendacji	189
Wybór cech	191
Regresja	195

Wybieranie odpowiednich cech	198
Wprowadzenie do uczenia maszynowego	199
Optyczne rozpoznawanie znaków	199
Budowa filtra spamu	200
Przewidywanie cen akcji	201
Powtórzenie	201
 11. Co dalej	203
 Drzewa	203
Odwrócone indeksy	206
Transformata Fouriera	207
Algorytmy równoległe	208
MapReduce	209
Do czego nadają się algorytmy rozproszone	209
Funkcja map	209
Funkcja reduce	210
Filtry Blooma i HyperLogLog	211
Filtry Blooma	212
HyperLogLog	213
Algorytmy SHA	213
Porównywanie plików	214
Sprawdzanie haseł	215
Locality-sensitive hashing	216
Wymiana kluczy Diffiego-Hellmana	217
Programowanie liniowe	218
Epilog	219
 Rozwiązania ćwiczeń	221
 Skorowidz	235





Przedmowa

Początkowo programowanie traktowałem jak hobby. Z książki *Visual Basic 6 for Dummies* nauczyłem się podstaw, które potem rozszerzałem, czytając inne podręczniki. Mimo to temat algorytmów pozostawał dla mnie niedostępny. Pamiętam, jak przeglądając spis treści pierwszej kupionej książki na temat algorytmów, myślałem sobie: „W końcu zrozumieć ten temat!”. Jednak tekst okazał się bardzo trudny, przez co po kilku tygodniach musiałem się poddać. Prostotę i elegancję algorytmów udało mi się zrozumieć dopiero wtedy, gdy zetknąłem się z moim pierwszym dobrym nauczycielem tego przedmiotu.

Kilka lat temu opublikowałem na swoim blogu pierwszy ilustrowany wpis. Jestem wzrokiem i bardzo lubię taki wizualny sposób prezentowania informacji. Potem napisałem jeszcze kilka tego typu wpisów na temat programowania funkcyjnego, systemu Git, uczenia maszynowego i współprzebieżności. A tak przy okazji: kiedy zaczynałem, byłem bardzo słabym pisarzem. Wyjaśnianie technicznych zawiłości nie jest łatwe. Trzeba długą myśleć, aby przytoczyć dobre przykłady, i poświęcić sporo czasu, aby dobrze objaśnić skomplikowane pojęcie. Dlatego najłatwiej trudniejsze kwestie omówić tylko pobieżnie. Myślałem, że świetnie mi idzie, dopóki jeden z moich wpisów nie zdobył większej popularności. Wówczas jeden ze współpracowników powiedział: „Przeczytałem Twój wpis, ale nadal tego nie rozumiem”. To mi uświadomiło, jak dużo muszę się jeszcze nauczyć o pisaniu.

Pewnego dnia zgłosiło się do mnie wydawnictwo Manning z pytaniem, czy chciałbym napisać ilustrowany podręcznik. Okazało się, że redaktorzy z tego wydawnictwa świetnie znają się na objaśnianiu technicznych zagadnień i to oni mi wy tłumaczyli, jak należy uczyć. Zdecydowałem się napisać tę książkę w jednym konkretnym celu. Chcę dobrze objaśnić trudny techniczny temat. Poza tym uznałem, że potrzebna jest przystępna książka o algorytmach. Moje umiejętności pisarskie znacznie się rozwinęły od czasu publikacji pierwszego wpisu na blogu, co pozwala mi żywić nadzieję, że książka będzie zrozumiała i przydatna dla czytelników.



Podziękowania

Wyrażam uznanie dla wydawnictwa Manning za umożliwienie mi napisania tej książki i pozostawienie dużej swobody twórczej. Dziękuję wydawcom Marjanowi Bace'owi i Mike'owi Stephensowi za wciągnięcie mnie na pokład, Bertowi Batesowi za kurs pisania oraz Jennifer Stout za gotowość do współpracy i bezcenną pomoc redaktorską. Dziękuję także członkom zespołu produkcyjnego wydawnictwa Manning: Kevinowi Sullivanowi, Mary Piergies, Tiffany Taylor, Leslie Haimes i wszystkim pozostałym. Dodatkowo na podziękowania zasłużyły osoby, które przeczytały rękopis i udzielili mi wielu porad. Oto one: Karen Bensdon, Rob Green, Michael Hamrah, Ozren Harlovic, Colin Hastie, Christopher Haupt, Chuck Henderson, Paweł Kozłowski, Amit Lamba, Jean-François Morin, Robert Morrison, Sankar Ramanathan, Sander Rossel, Doug Sparling oraz Damien White.

Dziękuję wszystkim, dzięki którym dotarłem do tego miejsca, a więc ludziom zajmującym się grą *Flaskhit*, którzy nauczyli mnie programować, i przyjacielom, którzy przeczytali różne rozdziały książki i udzielili mi wskazówek, pozwalając wypróbować na sobie różne objaśnienia. Zaliczają się do nich m.in. Ben Vinegar, Karl Puzon, Alex Manning, Esther Chan, Anish Bhatt, Michael Glass, Nikrad Mahdi, Charles Lee, Jared Friedman, Hema Manickavasagam, Hari Raja, Murali Gudipati, Srinivas Varadan i inni. Dziękuję też Gerry'emu Brady'emu za nauczenie mnie algorytmów. Kolejne wielkie podziękowania kieruję do badaczy algorytmów, a więc m.in. CLRS, Knutha i Stranga. Dosłownie stoję na ramionach gigantów.

Tato, mamo, Priyanko i cała pozostała rodzina: dziękuję za nieustanne wsparcie. Dziękuję także mojej żonie Maggie. Czeka nas jeszcze wiele przygód i nie wszystkie wiążą się z domowym przepisywaniem tekstu w piątkowe wieczory.

Na koniec dziękuję wszystkim czytelnikom, którzy postanowili dać szansę tej książce, oraz tym, którzy wyrazili opinię o niej na forum. Naprawdę pomogliście udoskonalić tę książkę.

O książce

Książka celowo została skonstruowana tak, aby była jak najłatwiejsza w odbiorze. Unikałem dużych skrótów myślowych. Każde nowe pojęcie szczegółowo objaśniałem od razu albo pisałem, gdzie je wyjaśnilem. Najważniejsze pojęcia opatrzyłem dodatkowo ćwiczeniami oraz opisałem na różne sposoby, aby można było zweryfikować swoje założenia i upewnić się, że wszystko jest w pełni zrozumiałe.

Stawiam na przykłady. Zamiast pokazywać gmatwaninę symboli, wolę wspomagać procesy myślowe za pomocą środków wizualnych. Poza tym uważam, że nauka jest najskuteczniejsza wtedy, gdy można sobie przypomnieć coś, co już się wie, a przykłady ułatwiają przypominanie różnych rzeczy. Aby więc np. zapamiętać różnicę między tablicami a listami powiązanymi (wyjaśnioną w rozdziale 2.), można pomyśleć o usadzeniu widzów w kinie. Ponadto jestem wzrokowcem, choć teraz pewnie nie jest to już żadnym odkryciem, zatem książka zawiera mnóstwo obrazków.

Treść tej książki została bardzo skrupulatnie wyselekcjonowana. Nie ma sensu pisać książki o wszystkich algorytmach sortowania — do tego służą Wikipedia i Khan Academy. Wszystkie opisane przeze mnie algorytmy mają praktyczne zastosowanie. Wielokrotnie używałem ich we własnej pracy i stanowią dobrą podstawę do zgłębiania bardziej zaawansowanych tematów. Międz lektury!

Plan książki

Trzy pierwsze rozdziały zawierają informacje podstawowe.

- **Rozdział 1.** W nim poznasz pierwszy algorytm o praktycznym zastosowaniu, czyli wyszukiwanie binarne. Dodatkowo nauczysz się analizować szybkość algorytmów za pomocą tzw. notacji dużego O. Metoda ta jest używana w całej książce do opisywania szybkości lub powolności omawianych algorytmów.
- **Rozdział 2.** Tu poznasz dwie podstawowe struktury danych, czyli tablice i listy powiązane. Struktur tych używam w całej książce, a poza tym na ich

podstawie tworzy się bardziej zaawansowane struktury danych, takie jak tablice skrótów (rozdział 5.).

- **Rozdział 3.** W nim dowiesz się, czym jest rekurencja, czyli poręczna technika stosowana w wielu algorytmach (np. szybkiego sortowania, który to algorytm opisany został w rozdziale 4.).

Z doświadczenia wiem, że początkującym najwięcej trudności sprawiają notacja dużego O i rekurencja, dlatego tematom tym poświęciłem nieco więcej miejsca niż innym zagadnieniom.

W pozostałej części książki opisuję algorytmy o szerokim zastosowaniu.

- **Techniki rozwiązywania zadań** zostały opisane w rozdziałach 4., 8. i 9. Jeśli natknesz się na jakiś problem i nie wiesz, jak go efektywnie rozwiązać, możesz spróbować techniki „dziel i rządź” (rozdział 4.) albo programowania dynamicznego (rozdział 9.). Ewentualnie możesz też uznać fakt, że nie istnieje efektywne rozwiązanie, i zadowolić się wynikiem w przybliżeniu, implementując algorytm zachłanny (rozdział 8.).
- **Tablice skrótów** zostały opisane w rozdziale 5. Tablica skrótów to niezmiernie przydatna struktura danych. Przechowuje się w niej pary klucz-wartość, takie jak np. nazwisko i adres e-mail albo nazwa użytkownika i hasło. Trudno przecenić ich wartość w programowaniu. Gdy podchodzę do jakiegoś nowego zadania, na początku rozważam dwie możliwości: „Czy da się tu użyć tablicy skrótów” oraz „Czy można to przedstawić w postaci grafu”.
- **Algorytmy grafów** zostały opisane w rozdziałach 6. i 7. Grafy służą do modelowania sieci społecznościowych, drogowych, neuronowych i wszystkich innych zbiorów połączeń. Najkrótszą drogę łączącą dwa punkty w sieci znajduje się za pomocą algorytmu wyszukiwania wszerz (rozdział 6.) i algorytmu Dijkstry (rozdział 7.). Przy ich użyciu można obliczyć, jak wiele dzieli dwie osoby albo znaleźć najkrótszą drogę do wybranego miejsca.
- **K najbliższych sąsiadów (KNN)** to algorytm opisany w rozdziale 10. Jest to prosty algorytm uczenia maszynowego. Przy użyciu KNN można utworzyć system rekomendacji, mechanizm optycznego rozpoznawania znaków albo system przewidywania wartości akcji giełdowych. Liczy się wszystko, co jest związane z przewidywaniem wartości („Naszym zdaniem Adit przymyta temu filmowi 4 gwiazdki”) lub klasyfikowaniem obiektów („Ta litera to Q”).
- **Dalszy rozwój.** W rozdziale 11. przedstawiam 10 algorytmów, z którymi warto się zapoznać.

Jak korzystać z tej książki

Kolejność tematów i treść tej książki zostały bardzo dokładnie przemyślane. Jeśli interesuje Cię tylko jeden temat, oczywiście możesz przejść od razu do odpowiedniego rozdziału. W pozostałych przypadkach najlepiej czytać rozdziały po kolejno, ponieważ każdy bazuje na poprzednich.

Gorąco zachęcam do samodzielnego uruchomienia przykładów kodu. Trudno przecenić wartość tego postępowania. Kod źródłowy można przepisać wprost z książki (albo pobrać archiwum z serwera FTP pod adresem <ftp://ftp.helion.pl/przyklady/algoip.zip>), a następnie uruchomić, aby zobaczyć, jak działa. Jeśli to zrobisz, nauczysz się znacznie więcej.

Polecam też wykonanie wszystkich ćwiczeń. Są one krótkie — w większości przypadków wystarczy parę minut na ich wykonanie, a tylko nieliczne wymagają poświęcenia około 5 – 10 minut. Ćwiczenia naprowadzą na właściwe tory myślowe i pozwolą zorientować się, że czegoś dobrze nie rozumiesz, zanim całkiem się pogubisz.

Kto powinien przeczytać tę książkę

Książka ta jest przeznaczona dla wszystkich tych, którzy znają podstawy programowania i chcą zrozumieć algorytmy. Może masz do rozwiązania jakiś problem programistyczny i szukasz odpowiedniego algorytmu? A może ktoś chce zrozumieć, do czego algorytmy mogą być przydatne? Oto krótka i niepełna lista grup osób, które mogą najbardziej skorzystać na lekturze tej książki:

- programiści amatorzy;
- studenci biorący udział w warsztatach programistycznych;
- absolwenci informatyki, którzy potrzebują odświeżenia wiadomości;
- fizycy, matematycy i absolwenci innych kierunków interesujący się programowaniem.

Konwencje typograficzne i pobieranie kodu

Wszystkie przykłady przedstawione w tej książce napisano w języku Python 2.7. Kod źródłowy zawsze jest drukowany czcionką o stałej szerokości znaków, aby odróżniał się od zwykłego tekstu. Niektóre listingi zawierają komentarze zwracające uwagę na ważne koncepcje.

Archiwum z plikami zawierającymi przykładowy kod źródłowy można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/algoip.zip>.

Uważam, że nauka jest najskuteczniejsza, gdy sprawia przyjemność. Dlatego życzę miłej zabawy oraz zachęcam do uruchomienia przykładów!

O autorze

Aditya Bhargava jest programistą w Etsy, internetowym rynku sprzedaje ręcznie robionych wyrobów. Posiada tytuł magistra informatyki University of Chicago i prowadzi popularny blog technologiczny pod adresem <adit.io>.



W tym rozdziale:

- zdobędziesz podstawowe wiadomości potrzebne do zrozumienia dalszej części książki,
- napiszesz swój pierwszy algorytm sortowania (wyszukiwanie binarne),
- nauczysz się opisywać czas działania algorytmów (notacja dużego O),
- poznasz powszechnie stosowaną technikę projektowania algorytmów (rekurencję).

Wprowadzenie

Algorytm to zestaw instrukcji opisujących, jak wykonać pewne zadanie. Algorytmem można wprawdzie nazwać każdy fragment kodu źródłowego, ale w tej książce przedstawiam nieco ciekawszą perspektywę. Algorytmy opisane tutaj wybierałem pod kątem ich szybkości lub rodzaju rozwiązywanych problemów albo obu tych cech. Oto kilka ważnych informacji.

- W rozdziale 1. opisuję wyszukiwanie binarne i pokazuję, jak algorytm może przyspieszyć działanie programu. W jednym z przykładów liczba czynności, jakie należy wykonać, została zredukowana z czterech miliardów do zaledwie 32!

- Urządzenia GPS najkrótszą drogę do celu obliczają przy użyciu algorytmów grafów (opisanych w rozdziałach 6., 7. i 8.).
- Przy użyciu technik programowania dynamicznego (opisanych w rozdziale 9.) można napisać algorytm sztucznej inteligencji grający w szachy.

W każdym przypadku opisuję algorytm i podaję przykład jego zastosowania. Następnie omawiam czas działania, używając notacji wielkiego O. Na koniec podpowiadam, jakie jeszcze inne zadania można rozwiązać za pomocą danego algorytmu.

Czego nauczysz się o wydajności

Dobra wiadomość jest taka, że w Twoim ulubionym języku programowania prawdopodobnie dostępna jest implementacja każdego algorytmu opisanego w tej książce, zatem nie musisz wszystkiego pisać samodzielnie! Jednak cała ta pomoc jest bezużyteczna dla kogoś, kto nie rozumie, jakie są ich mocne i słabe strony. W książce tej nauczysz się porównywać różne algorytmy. Czy w danym przypadku lepiej wybrać sortowanie przez scalanie, czy sortowanie szybkie? Czy lepiej użyć tablicy, czy listy? Sam wybór struktury danych może być szalenie ważny dla wydajności.

Czego nauczysz się o rozwiązywaniu problemów

Poznasz techniki rozwiązywania zadań, które do tej pory mogły być poza Twoim zasięgiem. Oto przykłady.

- Jeśli lubisz tworzyć gry wideo, możesz napisać system SI (sztucznej inteligencji), który z wykorzystaniem algorytmów grafowych będzie śledził użytkownika na planszy.
- Dowiesz się, jak utworzyć system rekomendacji za pomocą algorytmu k najbliższych sąsiadów.
- Niektórych problemów nie da się rozwiązać w zadowalającym czasie! W części poświęconej problemom NP-zupełnym nauczę Cię rozpoznawać te problemy i wybierać algorytmy, które pozwolą uzyskać przybliżone rozwiązanie.

Mówiąc bardziej ogólnie, po przestudiowaniu tej książki będziesz znać niektóre z najszerzej wykorzystywanych algorytmów programistycznych. Następnie możesz bardziej szczegółowo zgłębić wybraną grupę algorytmów, np. SI, baz danych itd. Ewentualnie możesz w pracy zająć się poważniejszymi zadaniami.

Co musisz wiedzieć

Przystępując do lektury tej książki, musisz znać podstawy algebra. Weźmy np. funkcję $f(x) = 2x$. Wiesz, ile wynosi $f(5)$? Jeśli Twoja odpowiedź brzmiała 10, tzn. że wiesz wystarczająco dużo.

Ponadto studiowanie tego rozdziału (i całej książki) będzie łatwiejsze dla tych, którzy znają przynajmniej jeden język programowania. Wszystkie przykłady są napisane w Pythonie. Jeśli więc nie znasz jeszcze żadnego języka programowania i chcesz się jakiegoś nauczyć, wybierz właśnie Python, który jest świetnym wyborem dla początkujących. Jeżeli znasz inny język, np. Ruby, to też sobie poradzisz.

Wyszukiwanie binarne

Wyobraź sobie, że szukasz w książce telefonicznej (co za staroświecka metoda!) nazwiska na literę K. Choć możesz zacząć szukanie od początku i przewracać kartki, aż dojdziesz do litery K, prawdopodobnie tego nie zrobisz, tylko od razu otworzysz książkę na środkowej stronie, ponieważ wiesz, że litera K znajduje się mniej więcej w środku alfabetu.

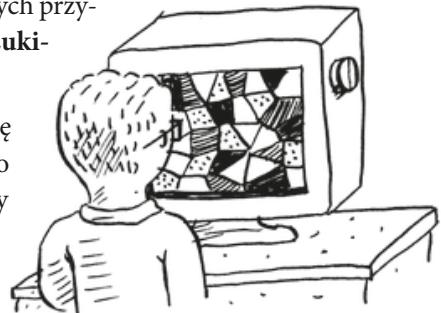
Albo pomyśl sobie, że szukasz w słowniku słowa na literę O. Tym razem również od razu otworzysz książkę na dalszej stronie.

A teraz wyobraź sobie, że logujesz się na Facebooku. Gdy to robisz, Facebook musi sprawdzić, czy na pewno masz konto w tym portalu. W związku z tym musi poszukać Twojej nazwy użytkownika w swojej bazie danych. Powiedzmy, że Twoja nazwa użytkownika to *karlmageddon*. Serwis może zacząć szukanie tej nazwy od litery A, ale bardziej sensowne wydaje się rozpoczęcie gdzieś w środku.

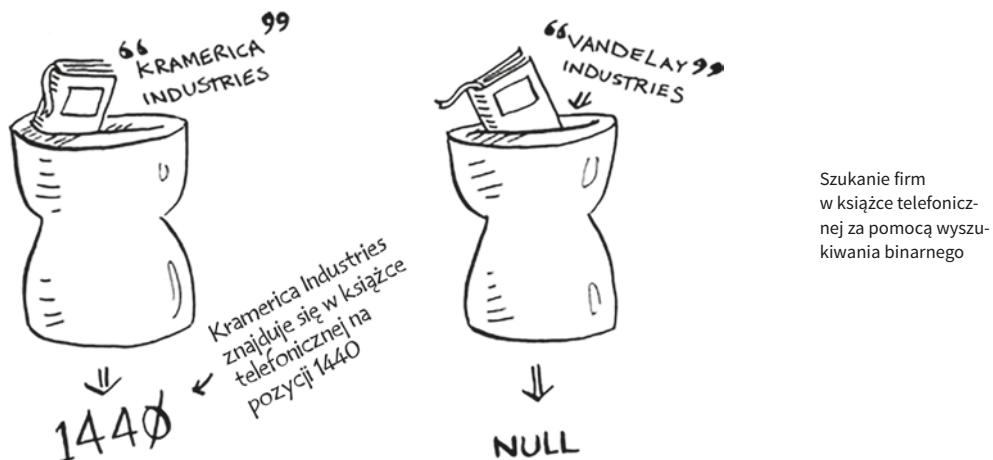
To jest problem wyszukiwania i we wszystkich wymienionych przypadkach należy użyć tego samego algorytmu, czyli **wyszukiwania binarnego**.

Wyszukiwanie binarne to algorytm. Na wejście podaje się posortowaną listę elementów (dalej wyjaśniam, dlaczego lista musi być posortowana). Jeśli lista ta zawiera szukany element, algorytm zwraca informację o jego położeniu.

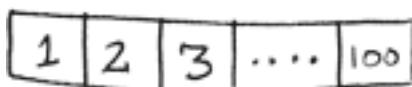
W przeciwnym przypadku zwracana jest wartość `null`.



Oto przykład.

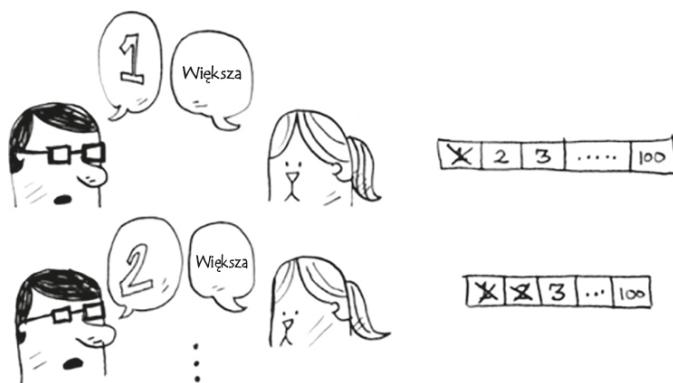


Oto przykład działania algorytmu wyszukiwania binarnego. Myślę o liczbie z przedziału od 1 do 100.



Musisz zgadnąć, o jakiej liczbie myślę w jak najmniejszej liczbie prób. Za każdym razem, gdy wymienisz jakąś liczbę, powiem Ci, czy jest większa, mniejsza, czy taka sama jak moja.

Powiedzmy, że zaczynasz zgadywanie w taki sposób: 1, 2, 3, 4... Tak by to wyglądało.





Złe podejście
do zgadywania liczb

To jest **wyszukiwanie proste** (choć chyba lepsza byłaby nazwa **wyszukiwanie głupie**). Przy każdej próbie eliminuje się tylko jedną liczbę. Gdybym pomyślał o liczbie 99, do jej odgadnięcia trzeba by aż 99 prób!

Lepszy sposób wyszukiwania

Oto lepsza metoda. Zaczniemy od 50.



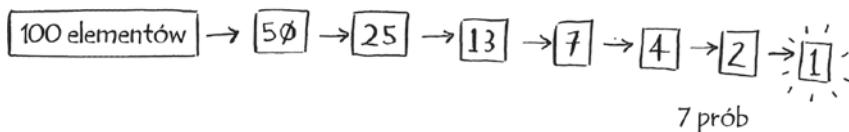
Większa, ale właśnie wyeliminowałeś *połowę* liczb! Wiesz już, że moja liczba jest większa od wszystkich liczb z przedziału od 1 do 50. Następny strzał to 75.



Tym razem moja liczba jest mniejsza, ale znów pozbyłeś się połowy zbioru! *Wyszukiwanie binarne* polega na zgadywaniu średkowej liczby w celu wyeliminowania w każdej próbie połowy pozostałych liczb. Następny strzał to liczba 63 (w połowie drogi między 50 i 75).



To jest właśnie wyszukiwanie binarne. Gratuluję poznania pierwszego algorytmu! Oto ile liczb można odrzucić w każdej próbie.



Eliminacja połowy liczb w każdej próbie wyszukiwania binarnego

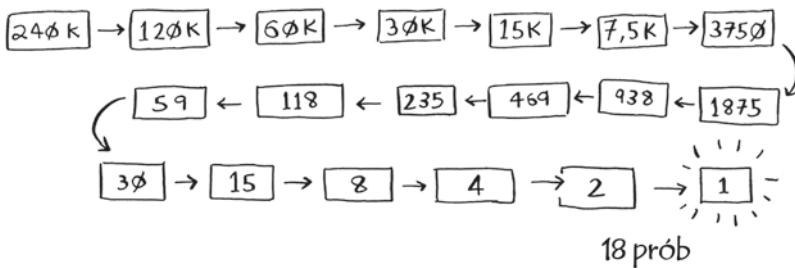
Kiedy w każdej próbie będzie eliminowana tak duża część przeszukiwanego zbioru, to bez względu na to, o jakiej liczbie bym nie pomyślał, zawsze możesz ją odgadnąć najwyżej w siedmiu próbach!

Powiedzmy, że szukamy hasła w słowniku, który zawiera 240 000 słów. Jak myślisz, ile razy trzeba będzie zgadywać w *najgorszym przypadku* przy zastosowaniu każdego z opisanych algorytmów?

Wyszukiwanie proste: _____ kroków

Wyszukiwanie binarne: _____ kroków

Gdyby szukane hasło było ostatnim słowem w słowniku, przy zastosowaniu algorytmu wyszukiwania prostego trzeba by wykonać 240 000 prób. Natomiast w wyszukiwaniu binarnym w każdej próbie eliminuje się połowę przeszukiwanego zbioru słów, aż pozostanie tylko jedno.



A zatem wyszukiwanie binarne wymaga wykonania 18 prób — spora różnica! Ogólnie rzecz biorąc, dla każdej listy n elementów wyszukiwanie binarne wymaga w najgorszym przypadku $\log_2 n$ prób, podczas gdy wyszukiwanie proste wymaga n prób.

Logarytmy

Możesz nie pamiętać, czym są logarytmy, ale na pewno wiesz, co to jest potęgowanie. Zapis $\log_{10} 100$ jest jak pytanie: „Ile dziesiątek trzeba przez siebie pomnożyć, aby otrzymać 100?”. Odpowiedź wynosi $2: 10 \cdot 10 = 100$. W związku z tym $\log_{10} 100$ wynosi 2. Innymi słowy, logarytmy można interpretować jako odwrotność potęgowania.

$$\begin{array}{rcl} 10^2 = 100 & \leftrightarrow & \log_{10} 100 = 2 \\ \hline 10^3 = 1000 & \leftrightarrow & \log_{10} 1000 = 3 \\ \hline 2^3 = 8 & \leftrightarrow & \log_2 8 = 3 \\ \hline 2^4 = 16 & \leftrightarrow & \log_2 16 = 4 \\ \hline 2^5 = 32 & \leftrightarrow & \log_2 32 = 5 \end{array}$$

Logarytmy są
odwrotnością
potęgowania

Kiedy w książce tej jest mowa o czasie wykonywania logarytmu wyrażonym w notacji dużego O (objaśnionej nieco dalej), \log zawsze oznacza \log_2 . Przy szukaniu elementu za pomocą algorytmu szukania prostego w najgorszym przypadku może być konieczne sprawdzenie wszystkich elementów. Jeśli więc lista zawiera 8 elementów, maksymalnie trzeba by obejrzeć 8 elementów. W wyszukiwaniu binarnym w najgorszym przypadku należy sprawdzić $\log n$ elementów. Dla listy 8 elementów mielibyśmy zatem wynik $\log 8 == 3$, ponieważ $2^3 == 8$. Innymi słowy, w przypadku listy 8 elementów należy sprawdzić maksymalnie 3 liczby. Gdyby lista zawierała 1024 elementy, to $\log 1024 == 10$, ponieważ $2^{10} == 1024$. Aby więc znaleźć liczbę na liście 1024 liczb, w najgorszym przypadku należy wykonać 10 prób.

Uwaga

W książce tej bardzo często wyrażam czas działania przy użyciu logarytmów, więc koniecznie dokładnie zrozum to pojęcie. Jeśli sprawia Ci ono trudności, możesz obejrzeć świetny film na ten temat w Khan Academy ([khanacademy.org](https://www.khanacademy.org)).

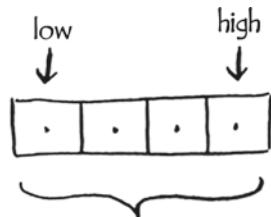
Uwaga

Wyszukiwanie binarne można stosować tylko wtedy, kiedy lista jest posortowana. Księga telefoniczna zawiera np. listę nazwisk posortowaną w kolejności alfabetycznej, więc można w niej szukać nazwisk za pomocą wyszukiwania binarnego. A co by się stało, gdyby nazwiska nie były posortowane?

Zobaczmy, jak napisać algorytm wyszukiwania binarnego w Pythonie. W przedstawionych tu przykładach używam tablic. Nie przejmuj się, jeśli nie wiesz, czym są tablice, ponieważ ich szczegółowy opis zamieściłem w następnym rozdziale. Na razie wystarczy tylko zapamiętać, że w pamięci komputera szereg elementów można zapisać w kolejnych komórkach i taki szereg nazywa się tablicą. Komórki mają numery zaczynające się od 0: pierwsza znajduje się na pozycji nr 0, druga na pozycji nr 1, trzecia na pozycji nr 2 itd.

Funkcja `binary_search` pobiera posortowaną tablicę i element. Jeśli element ten występuje w tablicy, funkcja zwraca numer jego pozycji. Będziesz śledzić, którą część tablicy trzeba przeszukać. Na początek zdefiniujemy naszą tablicę.

```
low = 0
high = len(list) - 1
```



To są wszystkie
liczby, które
będziemy przeszukiwać

W każdej próbie sprawdzamy element środkowy.

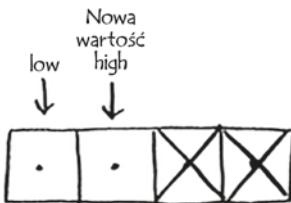
```
mid = (low + high) / 2
```

Python automatycznie zaokrągli wartość mid w dół,
jeśli wynik low + high będzie nieparzysty.

```
guess = list[mid]
```

Jeśli wartość `guess` jest za mała, odpowiednio zmieniamy `low`.

```
if guess < item:
    low = mid + 1
```



A jeśli wartość `guess` jest za duża, to zmieniamy wartość `high`. Oto kompletny kod.

```
def binary_search(list, item):
    low = 0           Za pomocą low i high kontrolujemy, która
    high = len(list)-1   część listy jest przeszukiwana.
    while low <= high:   Dopóki obszar poszukiwań nie został zawię-
        mid = (low + high)  żony do jednego elementu...
        guess = list[mid]   ...wybieramy środkowy element.
        if guess == item:   Znaleziono element.
            return mid
        if guess > item:   Strzelono za dużą liczbę.
            high = mid - 1
        else:               Strzelono za małą liczbę.
            low = mid + 1
    return None          Nie ma takiego elementu.
                        Przetestujmy to!
```

`my_list = [1, 3, 5, 7, 9]` Pamiętaj, że numerowanie
w listach zaczyna się od 0. Druga
komórka ma indeks 1.

```
print binary_search(my_list, 3) # => 1
print binary_search(my_list, -1) # => None Wartość None w Pythonie oznacza  
nic, czyli wskazuje, że elementu  
nie znaleziono.
```

ĆWICZENIA

- 1.1.** Powiedzmy, że mamy posortowaną listę 128 nazwisk i chcemy ją przeszukać za pomocą algorytmu wyszukiwania binarnego. Ile maksymalnie prób zgadywania będzie trzeba wykonać?
- 1.2.** Ile maksymalnie prób zgadywania trzeba wykonać, jeśli podwoi się liczbę elementów w liście?

Czas wykonywania

W opisie każdego algorytmu zamieszczam też informację na temat czasu wykonywania. Generalnie zawsze należy wybierać najefektywniejszy algorytm, zapewniający optymalne wykorzystanie czasu lub miejsca w pamięci.

Wracamy do wyszukiwania binarnego. Ile czasu można oszczędzić przy jego zastosowaniu? Pierwsza metoda polegała na sprawdzaniu każdej liczby po kolej. Jeśli lista zawiera 100 liczb, znalezienie na niej elementu może wymagać wykonania maksymalnie 100 operacji. Gdyby lista ta zawierała cztery miliardy elementów, trzeba by wykonać cztery miliardy prób. A zatem maksymalna liczba strzałów jest równa liczbie elementów w liście. Nazywa się to **czasem liniowym**.

Z wyszukiwaniem binarnym jest inaczej. Jeśli lista zawiera 100 elementów, to aby znaleźć w niej element, maksymalnie należy wykonać 7 prób. Gdyby lista zawierała cztery miliardy elementów, maksymalna liczba strzałów wynosiłaby 32. Nieźle, prawda? Wyszukiwanie binarne charakteryzuje się **logarytmicznym czasem wykonywania**. Tabela obok przedstawia zwięzłe podsumowanie naszych ustaleń.

Wyszukiwanie proste	Wyszukiwanie binarne
100 elementów ↓ 100 prób	100 elementów ↓ 7 prób
4 000 000 000 elementów ↓ 4 000 000 000 prób	4 000 000 000 elementów ↓ 32 próby
$O(n)$	$O(\log n)$
Czas liniowy	 Wielkie oszczędności
	 Wielkie oszczędności
	Czasy wykonywania algorytmów sortujących
	 Czas logarytmiczny



Notacja dużego O

Notacja **dużego O** to specjalny sposób opisu szybkości działania algorytmów. A kogo to obchodzi? Otóż musisz wiedzieć, że bardzo często będziesz używać algorytmów opracowanych przez innych programistów i wówczas z przyjemnością zerkniesz na informację, jak szybko lub wolno one działają. W tym podrozdziale wyjaśniam zatem, czym jest notacja dużego O oraz przedstawiam przy jej użyciu niektóre najbardziej typowe czasy wykonania różnych algorytmów.

Czas wykonywania algorytmów rośnie w różnym tempie

Bartek pisze algorytm wyszukiwania dla agencji NASA. Algorytm ten będzie uruchamiany tuż przed lądowaniem rakiety na Księżycu, a jego zadaniem będzie pomóc w obliczeniu najlepszego miejsca do posadowienia statku.

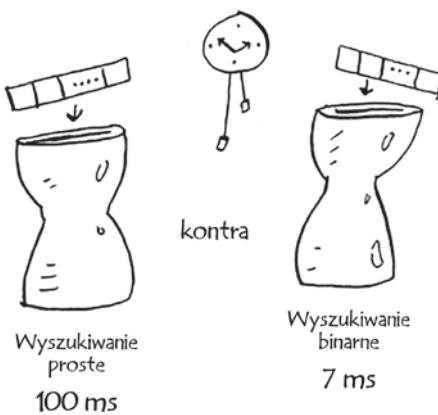
Jest to doskonały przykład tego, jak czas wykonywania dwóch algorytmów może rosnąć w różnym tempie. Bartek zastanawia się nad wyborem algorytmu wyszukiwania prostego i binarnego. Jego rozwiązanie musi być i szybkie, i dawać poprawne wyniki. Z jednej strony, wyszukiwanie binarne jest szybsze, a Bartek ma tylko 10 sekund na znalezienie miejsca do lądowania albo rakieta zejdzie z kursu. Z drugiej strony, proste wyszukiwanie jest łatwiejsze do zaimplementowania, więc występuje mniejsze ryzyko popełnienia błędu. A Bartek *bardzo* by nie chciał popełnić błędu w kodzie odpowiadającym za lądowanie rakiety! Aby mieć absolutną pewność o słuszności swojego wyboru, Bartek postanowił przetestować oba algorytmy na liście 100 elementów.

Powiedzmy, że sprawdzenie jednego elementu trwa milisekundę. Algorytm wyszukiwania prostego musi sprawdzić 100 elementów, więc wyszukiwanie zajmie nie więcej niż 100 ms. Algorytm

wyszukiwania binarnego musi sprawdzić maksymalnie 7 elementów (ponieważ $\log_2 100$ z grubsza wynosi 7), a więc czas działania nie będzie dłuższy niż 7 ms. Jednak w rzeczywistości lista może mieć nawet miliard elementów. Ile wówczas czasu zajmie wyszukiwanie proste, a ile binarne? Zastanów się dokładnie nad odpowiedzią na to pytanie, zanim przeczytasz następne akapity.

Bartek wykonał wyszukiwanie binarne w liście zawierającej miliard elementów i stwierdził, że zajęło mu to 30 ms

($\log_2 1\ 000\ 000\ 000$ wynosi mniej więcej 30). Pomyślał sobie: „Aha, 32 ms! Wyszukiwanie binarne jest około 15 razy szybsze od prostego, ponieważ wyszukiwanie proste w liście 100 elementów zajęło 100 ms, a wyszukiwanie binarne w tej samej liście trwało 7 ms. W związku z tym proste przeszukanie listy miliarda elementów powinno zajść $30 \cdot 15 = 450$ ms, prawda? To znacznie mniej niż graniczne 10 sekund”. Na podstawie tych przemyśleń Bartek wybiera wyszukiwanie proste. Czy podjął słuszną decyzję?



Porównanie czasów wykonywania wyszukiwania prostego i binarnego w liście 100 elementów

Nie. Bartek grubo się myli. I to nawet bardzo grubo. Czas przeszukiwania listy miliarda elementów za pomocą algorytmu wyszukiwania prostego wynosi miliard milisekund, czyli 11 dni! Sęka w tym, że czasy wykonywania algorytmów wyszukiwania prostego i binarnego *nie rosną w tym samym tempie!*

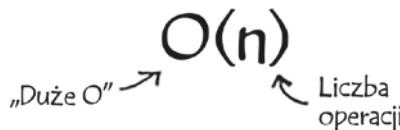
	Wyszukiwanie proste	Wyszukiwanie binarne	Czasy wykonywania rosną w różnym tempie!
100 elementów	100 ms	7 ms	
10 000 elementów	10 sekund	14 ms	
1 000 000 000 elementów	11 dni	32 ms	

Jak widać, zwiększanie liczby elementów do przeszukania w przypadku wyszukiwania binarnego tylko nieznacznie wydłuża czas wykonywania algorytmu. Natomiast w przypadku wyszukiwania prostego czas ten wydłuża się *radykalnie*. Zatem w miarę zwiększania się liczby elementów wyszukiwanie binarne nagle staje się znacznie szybsze od wyszukiwania prostego. Bartek myślał, że wyszukiwanie proste jest tylko 15 razy wolniejsze od binarnego, ale był w błędzie. Kiedy lista zawiera miliard elementów, ten drugi algorytm jest aż 33 miliony razy szybszy. Dlatego sama informacja, ile czasu dany algorytm był wykonywany, jest niewystarczająca. Dodatkowo trzeba wiedzieć, jak wydłuża się czas wykonywania wraz ze zwiększeniem liczby elementów. I tu właśnie staje się przydatna notacja dużego O.

Notacja dużego O informuje o tym, jak szybki jest algorytm. Powiedzmy np., że mamy listę o rozmiarze n . Wyszukiwanie proste musi sprawdzić każdy element po kolej, więc wykona n operacji. W notacji dużego O czas wykonywania tego algorytmu wyrazimy tak: $O(n)$. Gdzie są sekundy? Nie ma sekund — ta notacja nie wyraża szybkości w sekundach. *Notacja dużego O pozwala porównać liczbę operacji do wykonania.* Informuje, jak szybko rośnie czas wykonywania algorytmu.



Oto inny przykład. Aby sprawdzić listę o rozmiarze n , wyszukiwanie binarne musi wykonać $\log n$ operacji. Jak zatem wyrazić czas wykonywania tego algorytmu w notacji dużego O? Ano tak: $O(\log n)$. Ogólnie poszczególne elementy tego zapisu mają następujące znaczenie.



Poszczególne składniki zapisu w notacji dużego O

Z zapisu tego dowiadujemy się, ile operacji wykona dany algorytm. Nazwa notacja dużego O wzięła się z tego, że przed liczbą operacji stawia się literę „O” (brzmi jak żart, ale to prawda!).

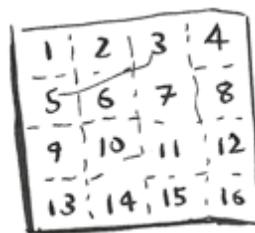
Przyjrzyjmy się kilku przykładom. Spróbuj określić czas wykonywania kilku następnych algorytmów.

Wizualizacja różnych czasów wykonywania

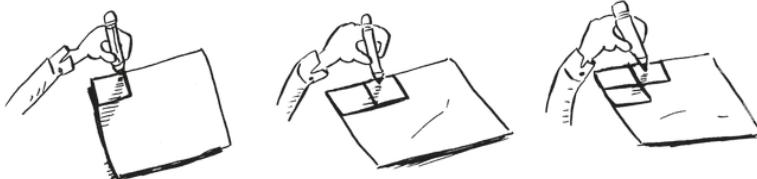
Obok przedstawiam praktyczny przykład metody, którą można zastosować w domu, jeśli ma się kawałek papieru i ołówek. Powiedzmy, że mamy narysować siatkę złożoną z 16 prostokątów.

Algorytm 1.

Jednym ze sposobów jest narysowanie 16 prostokątów, każdego osobno. Przypomnę, że notacja dużego O wyraża liczbę operacji. W tym przykładzie pojedynczą operacją jest narysowanie prostokąta. Do narysowania mamy 16 prostokątów. Ile operacji trzeba wykonać, aby narysować 16 takich prostokątów, po jednym na raz?



Znasz dobry algorytm do narysowania takiej siatki?



Rysowanie siatki po jednym prostokącie na raz

Narysowanie 16 prostokątów wymaga wykonania takiej samej liczby operacji. Jaki jest więc czas wykonywania tego algorytmu?

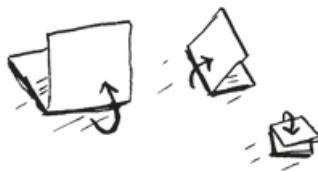
Algorytm 2.

Teraz wypróbuj inny algorytm. Zegnij kartkę.

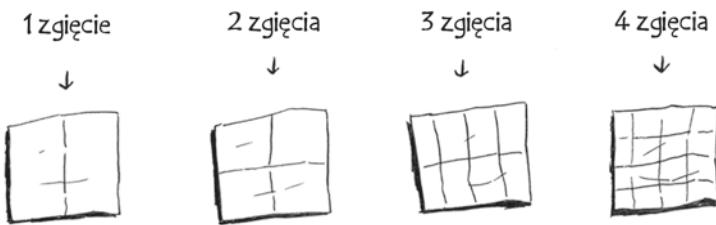


W tym przypadku zgięcie kartki na pół jest operacją, za pomocą której od razu utworzyliśmy dwa prostokąty!

Zegnij kartkę jeszcze raz, potem jeszcze raz, a potem jeszcze raz.



Teraz rozwiń kartkę i podziwiaj piękną siatkę prostokątów! Każde kolejne zgięcie podwaja liczbę kratek, dzięki czemu 16 prostokątów utworzyliśmy, wykonując tylko cztery operacje!



Rysowanie siatki w czterech krokach

Każde kolejne złożenie kartki powoduje podwojenie liczby kratek, dzięki czemu wystarczyły cztery operacje, aby narysować 16 kratek. Jaki zatem jest czas wykonywania tego algorytmu? Zanim przeczytasz następne akapity, zastanów się, jaki jest czas wykonywania obu opisanych do tej pory algorytmów.

Odpowiedzi: czas wykonywania pierwszego algorytmu to $O(n)$, a drugiego — $O(\log n)$.

Notacja dużego O określa czas działania w najgorszym przypadku

Powiedzmy, że za pomocą wyszukiwania prostego szukamy nazwiska w książce telefonicznej. Wiemy, że czas wykonywania tego algorytmu to $O(n)$, co oznacza, że w najgorszym przypadku będziemy musieli przejrzeć wszystkie pozycje w książce. W tym przypadku jednak szukamy nazwiska Adit, które znajduje się na pierwszym miejscu w naszej książce. Nie musielibyśmy więc przeszukiwać wszystkich pozycji, ponieważ szukany element znaleźliśmy już na pierwszej pozycji. Czy wykonywanie tego algorytmu zajęło $O(n)$ czasu? Czy też może czas jego wykonywania wyniósł $O(1)$, ponieważ szukany element trafiliśmy już w pierwszej próbie?

Algorytm wyszukiwania prostego nadal ma czas wykonywania $O(n)$. W tym przypadku od razu znaleźliśmy to, czego było nam potrzeba. To jest najlepszy przypadek. Jednak notacja dużego O zawsze opisuje *najgorszy przypadek*. Można więc powiedzieć, że w najgorszym przypadku będzie potrzeba przejrzeć wszystkie pozycje w książce telefonicznej. To jest czas wykonywania $O(n)$. Jest to rodzaj zapewnienia, że wyszukiwanie proste na pewno nie będzie trwać dłużej niż $O(n)$.

Uwaga

Oprócz czasu wykonywania w najgorszym przypadku, ważnym parametrem jest średni czas działania algorytmu. Szerzej te dwa parametry porównuję w rozdziale 4.

Kilka typowych czasów wykonywania

Poniżej przedstawiam listę pięciu czasów wykonywania wyrażonych w notacji dużego O, które jeszcze nie raz spotkasz w swojej pracy. Pierwszy jest najszybszy.

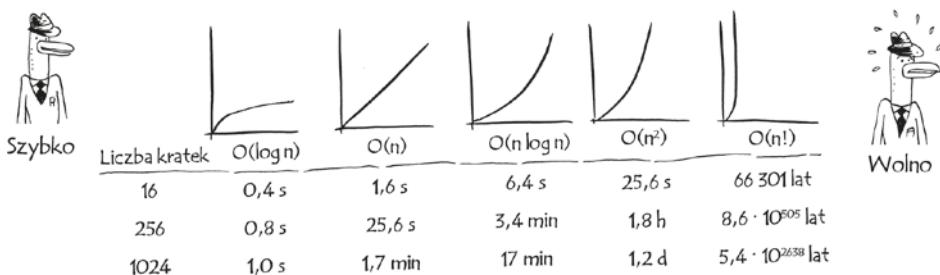
- $O(\log n)$ — zwany też **czasem logarytmicznym**. Przykład to wyszukiwanie binarne.
- $O(n)$ — zwany też **czasem liniowym**. Przykład to wyszukiwanie proste.
- $O(n \cdot \log n)$. Przykład to algorytm szybkiego sortowania (opisany w rozdziale 4.).
- $O(n^2)$. Przykład to wolny algorytm sortowania, np. sortowanie przez wybieranie (opisany w rozdziale 2.).

- $O(n!)$. Przykład to bardzo wolny algorytm sortowania, np. algorytm rozwiązuający problem komiwojażera (opisany w kolejnym punkcie!).

Powiedzmy, że jeszcze raz chcesz narysować siatkę składającą się z 16 kratek i do wyboru masz jeden z pięciu algorytmów. Jeśli wybierzesz pierwszy z nich, rysowanie siatki zajmie $O(\log n)$ czasu. Twój komputer jest w stanie wykonać 10 operacji na sekundę. Przy czasie wykonywania $O(\log n)$ narysowanie 16 kratek wymaga wykonania czterech operacji ($\log 16$ wynosi 4). W związku z tym narysowanie siatki zajmie Ci 0,4 sekundy. A gdyby trzeba było narysować 1024 kratki? Liczba operacji wynosiłaby wówczas $\log 1024$ czyli 10, a więc rysowanie siatki złożonej z 1024 prostokątów zajęłoby sekundę. Te obliczenia dotyczą pierwszego z wymienionych algorytmów.

Drugi algorytm jest wolniejszy, ponieważ czas jego wykonywania wynosi $O(n)$. Zatem narysowanie 16 kratek wymaga wykonania 16 operacji, a narysowanie 1024 prostokątów wymaga wykonania 1024 operacji. Ile to czasu w sekundach?

Poniższe wykresy przedstawiają czas rysowania siatki przy użyciu wszystkich opisanych algorytmów, zaczynając od najszybszego.



Spotyka się też inne czasy wykonywania, ale te wymienione wyżej występują najczęściej.

Oczywiście to wszystko jest uproszczone, ponieważ w rzeczywistości nie można wykonać takiej prostej konwersji czasu wykonywania wyrażonego w notacji dużego O na liczbę operacji, ale na razie to wystarczy. Do notacji dużego O wrócę jeszcze w rozdziale 4., gdy będziesz już znał trochę więcej algorytmów. Teraz przede wszystkim zapamiętaj następujące informacje.

- Szybkość wykonywania algorytmów nie wyraża się w sekundach, tylko w tempie wzrostu liczby operacji.
- Omawiając szybkość działania algorytmu, podaje się, jak szybko rośnie czas wykonywania wraz ze zwiększeniem rozmiaru zbioru wejściowego.

- Czas wykonywania algorytmów wyraża się za pomocą notacji dużego O.
- Algorytm o czasie wykonywania $O(\log n)$ jest szybszy niż $O(n)$, a im większy zbiór danych do przeszukania, tym większa robi się różnica.

ĆWICZENIA

Przedstaw czas wykonywania w każdym z opisanych poniżej przypadków za pomocą notacji dużego O.

- 1.3.** Dane jest nazwisko i trzeba znaleźć numer telefonu osoby o tym nazwisku w książce telefonicznej.
- 1.4.** Dany jest numer telefonu i trzeba znaleźć nazwisko właściciela tego numeru w książce telefonicznej. (Podpowiedź: musisz przeszukać całą książkę!).
- 1.5.** Chcesz przeczytać numery wszystkich osób w książce telefonicznej.
- 1.6.** Chcesz przeczytać numery tylko osób o nazwiskach na A. (Tu jest pułapka! Trzeba posłużyć się wiedzą przedstawioną bardziej szczegółowo w rozdziale 4. Przeczytaj odpowiedź — może Cię zaskoczyć!).

Problem komiwojażera

Czytając poprzedni punkt, niektórzy mogli sobie pomyśleć: „Nie ma szans, abyem kiedykolwiek natrafił na algorytm o czasie wykonywania $O(n!)$ ”. Jeśli należysz do tych osób, pozwól, że wyprowadzę Cię z błędu! Oto przykład algorytmu charakteryzującego się bardzo słabym czasem wykonywania. W informatyce problem ten jest powszechnie znany, ponieważ cechuje go właśnie wyjątkowe tempo wzrostu poziomu złożoności i wiele bardzo bystrych osób twierdzi, że nie da się z tym nic zrobić. Jest to tzw. **problem podrózującego komiwojażera** (ang. *traveling salesman problem*).

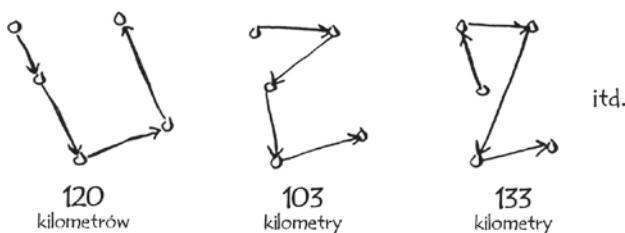


Mamy komiwojażera.

Komiwojażer ten musi zawitać do pięciu miast.



Nasz sprzedawca, nazwijmy go Opus, chce odwiedzić każde z pięciu miast, pokonując jak najmniejszy dystans. Oto jeden ze sposobów na rozwiązywanie tego zadania: sprawdzić wszystkie możliwe kolejności odwiedzania miast.



Sprzedawca sumuje odległości, a następnie wybiera najkrótszą drogę. Dla pięciu miast istnieje 120 permutacji, a więc rozwiązywanie zadania dla pięciu miast wymaga wykonania 120 operacji. Dla 6 miast liczba operacji wynosi już 720 (istnieje 720 permutacji), a dla 7 miast ta liczba sięga już 5040!

Miasta	Operacje
6	720
7	5040
8	40 320
...	...
15	1307 674 368 000
...	...
30	265 252 859 812 191 058 636 308 480 000 000

Liczba operacji
radykalnie rośnie

A to uogólnienie: dla n elementów obliczenie wyniku wymaga wykonania $n!$ („ n silnia”) operacji, a więc jest to algorytm charakteryzujący się **czasem wykonywania** $O(n!)$. Rozwiązywanie problemu dla jakiejkolwiek liczby elementów, nie licząc paru najmniejszych, zajmuje bardzo dużo czasu. Gdyby miast było więcej niż 100, obliczenia trwałyby tak długo, że szybciej doszłoby do śmierci naszego Słońca.

Ten algorytm jest straszny! Opus powinien wybrać jakiś inny, prawda? Jednak nie może tego zrobić, ponieważ jest to właśnie jeden z nieroziwiązanych problemów informatyki. Nikt nie zna szybkiego algorytmu rozwiązującego to zadanie, a bardzo mądrzy ludzie uważają, że opracowanie takiego algorytmu jest *niemożliwe*. Dlatego najlepszym wyjściem z tej sytuacji jest zadowolenie się wynikiem przybliżonym — więcej na ten temat piszę w rozdziale 10.

I jeszcze jedna uwaga: zaawansowani czytelnicy mogą zainteresować się binarnymi drzewami poszukiwań! Opisałem je krótko w ostatnim rozdziale.

Powtórzenie

- Wyszukiwanie binarne jest znacznie szybsze od wyszukiwania prostego.
- $O(\log n)$ oznacza szybszy algorytm niż $O(n)$ i im większy zbiór do przeszukania, tym większa robi się różnica prędkości.
- Szybkości algorytmów nie mierzy się w sekundach.
- Szybkość algorytmów określa się, podając tempo zwiększenia się ilości pracy.
- Szybkość algorytmów przedstawia się za pomocą notacji dużego O .





W tym rozdziale:

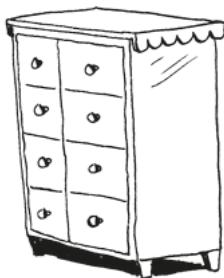
- poznasz tablice i listy powiązane, czyli dwie najbardziej podstawowe struktury danych, które stosuje się absolutnie wszędzie. Tablic użyłem już w rozdziale 1. i będę z nich korzystać w prawie wszystkich pozostałych rozdziałach do końca tej książki. Tablice to absolutna podstawa, więc nie zaniedbij ich! Mimo to czasami lepiej posłużyć się listą powiązaną, dlatego w rozdziale tym zamieszczam zestawienie zalet i wad obu tych struktur, aby umożliwić Ci wybór odpowiedniej z nich do użycia z każdym algorytmem.
- poznasz pierwszy algorytm sortowania. Wiele innych algorytmów działa tylko na zbiorach danych, które są posortowane. Pamiętasz wyszukiwanie binarne? W ten sposób można przeszukiwać tylko posortowane listy elementów. W tym rozdziale poznasz algorytm sortowania przez wybieranie. Większość języków programowania ma wbudowany jakiś algorytm sortowania, więc rzadko się zdarza, aby trzeba było go implementować we własnym zakresie. Jednak sortowanie przez wybieranie to jeden z etapów nadrode do sortowania szybkiego, które opisuję w następnym rozdziale. Sortowanie szybkie to bardzo ważny algorytm, którego zasadę działania łatwiej zrozumieć, gdy zna się już jakiś inny algorytm sortowania.

Co trzeba wiedzieć

Aby zrozumieć przedstawione w tym rozdziale analizy wydajności, należy znać notację dużego O i logarytmy. Jeśli masz braki w tych dwóch dziedzinach, zachęcam do przeczytania rozdziału 1. Notacją dużego O posługuje się już do końca tej książki.

Jak działa pamięć

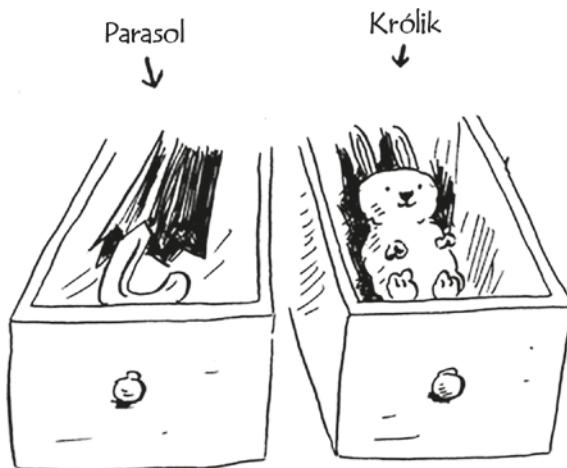
Wyobraź sobie, że idziesz na przedstawienie i chcesz sprawdzić swoje rzeczy. Do dyspozycji masz komodę.



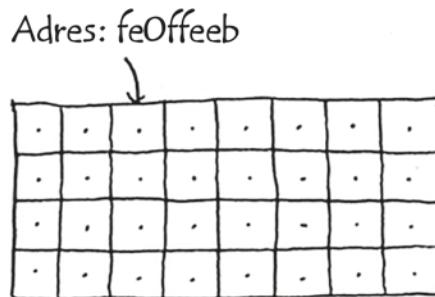
Każda szuflada pomieści jeden przedmiot. Chcesz przechować dwie rzeczy, więc prosisz o udostępnienie dwóch szuflad.



Do wskazanych szuflad wkładasz swoje rzeczy.



I jesteś gotowy na przedstawienie! Tak z grubsza działa też pamięć w komputerze, który można porównać do gigantycznej komody z szufladami posiadającymi adresy.



Ciąg znaków fe0ffe0b to adres komórki w pamięci.

Gdy chcesz coś zapisać w komórce pamięci, prosisz komputer o udostępnienie miejsca, a ten przekazuje Ci adres, pod którym możesz składować swoje dane. Jeśli chcesz zapisać kilka pozycji, możesz to zrobić na dwa podstawowe sposoby: wykorzystując tablicę lub listę. Najpierw omawiam tablice, a potem listy i dodatkowo przedstawiam ich zalety i wady. Nie istnieje jedyny właściwy sposób przechowywania danych w każdej sytuacji i dlatego trzeba wiedzieć, czym różnią się od siebie poszczególne struktury.

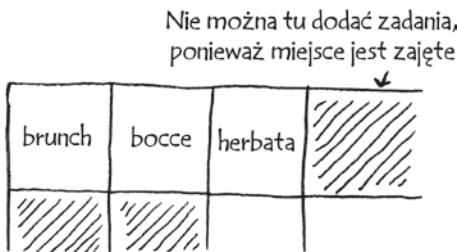
Tablice i listy powiązane

Czasami trzeba zapisać listę elementów w pamięci. Powiedzmy, że piszemy aplikację do planowania codziennych zadań. Zadania te powinniśmy zapisywać w pamięci właśnie w postaci listy.

Lepiej użyć tablicy czy listy powiązanej? Najpierw użyjemy tablicy, ponieważ łatwiej będzie ją zrozumieć. Elementy tablicy są przechowywane w kolejnych komórkach pamięci (jeden obok drugiego).



A teraz wyobraź sobie, że chcesz dodać jedno zadanie. Problem w tym, że następna szufladka jest zajęta przez kogoś innego!



To tak, jakbyśmy poszli ze znajomymi na film i znaleźliśmy sobie odpowiednie miejsca, a tu nagle dołącza jeszcze jeden znajomy, dla którego nie ma już miejsca obok nas. Wówczas wszyscy muszą się przenieść do innego rzędu, w którym jest wystarczająco dużo wolnych foteli obok siebie. W takim przypadku musimy poprosić komputer o inny fragment pamięci, w którym zmieszcza się wszystkie nasze zadania, a następnie zapisać je tam.

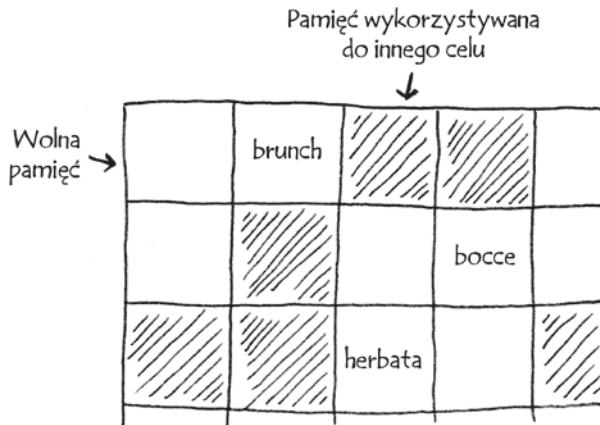
A jeśli przyjdzie kolejny znajomy, to znowu zabraknie nam miejsca i będzieemy musieli przenieść się po raz drugi! Co za porażka. Podobnie niewygodne może być dodawanie nowych elementów do tablicy. Gdyby przy dodawaniu każdego elementu brakowało miejsca i trzeba było się przenosić, operacje te byłyby bardzo nieefektywne. W tej sytuacji jednym z najprostszym rozwiązań jest „zajmowanie miejsc” — jeśli nawet aktualnie na liście zadań znajdują się tylko trzy pozycje, można poprosić komputer o przydzielenie dziesięciu miejsc, tak na wszelki wypadek. To bardzo dobre rozwiązań, choć też niepozbawione wad.

- Dodatkowe miejsce, o które poprosiliśmy, może się nam nie przydać i wówczas tylko zmarnujemy pamięć. Mimo że my go nie używamy, nikt inny też nie będzie mógł z niego skorzystać.
- Lista może się rozrosnąć powyżej 10 elementów, a wówczas i tak trzeba będzie się przenieść.

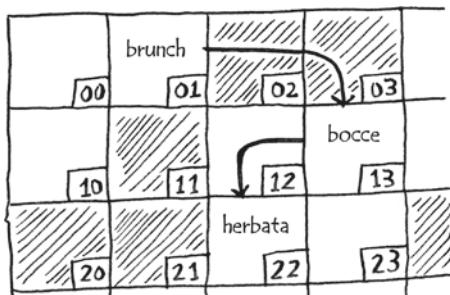
Zatem jest to wyjście dobre, ale nie idealne. Problem ten dobrze rozwiążają natomiast listy powiązane.

Listy powiązane

Elementy listy powiązanej mogą być rozrzucone po całej pamięci komputera.



Każdy element zawiera adres następnego elementu na liście. W ten sposób grupa losowo wybranych adresów w pamięci została ze sobą powiązana.



Powiązane adresy
w pamięci

Trochę przypomina to szukanie skarbu. Idziemy pod pierwszy adres i dowiadujemy się tam, np. że: „Następny element znajduje się pod adresem 123”. Idziemy więc pod adres 123, a tam widnieje informacja: „Następny element znajduje się pod adresem 847” itd. Dodawanie elementów do listy powiązanej jest łatwe; wystarczy zapisać dane w dowolnym miejscu, a następnie w poprzednim elemencie zapisać adres do tej komórki.

Elementów listy powiązanej nigdy nie trzeba przenosić. Dodatkowo wyeliminowany jest jeszcze inny problem. Powiedzmy, że idziemy na popularny film w sześciu osobowej grupie. Szukamy miejsca do siedzenia, ale sala kinowa jest zapełniona po brzegi i nigdzie nie ma sześciu wolnych miejsc obok siebie. Czasami coś takiego zdarza się też tablicom. Powiedzmy, że chcemy znaleźć 10 000 wolnych miejsc dla tablicy. W pamięci jest tyle wolnych komórek, ale nie w jednym rzędzie. W efekcie nie mamy gdzie zapisać naszej tablicy! Gdybyśmy zachowywali się, jak lista powiązana, to powiedzielibyśmy: „Rozdzielimy się i obejrzymy film oddzielnie”. Jeśli w pamięci jest wystarczająco dużo miejsca, to zawsze zapiszemy w niej listę powiązaną.

Skoro więc listy powiązane tak bardzo przewyższają tablice pod względem efektywności wstawiania elementów, w czym są dobre tablice?

Tablice

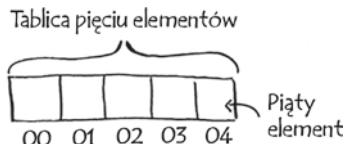
Na stronach internetowych zawierających listy „10 najlepszych” stosowane są wątpliwe etycznie sztuczki mające wygenerować jak najwięcej odsłon. Zamiast wyświetlić wszystko na jednej stronie, autor umieszcza każdy punkt listy na osobnej stronie, tak aby użytkownik podczas przeglądania musiał kliknąć przycisk *Dalej*. Przykładowo listy 10 najgorszych złoczyńców z telewizji nie zobaczysz w całości na jednej stronie, tylko najpierw zostanie pokazany nr 10 (Newman), a potem będziesz musiał kliknąć przycisk *Dalej*, aby zobaczyć następnych i dojść do nr 1 (Gustavo Fring). W ten sposób właściciele



serwisów internetowych zdobywają dodatkowe odsłony stron, na których mogą wyświetlać reklamy, choć dla użytkownika klikanie tyle razy jednego przycisku jest nudne. O wiele lepiej byłoby, gdyby cała lista znajdowała się na jednej stronie i można by kliknąć nazwisko każdej osoby w celu przeczytania dodatkowych informacji.

Podobny problem dotyczy list powiązanych. Powiedzmy, że chcemy odczytać ostatni element takiej listy. Nie możemy tak po prostu tego zrobić, ponieważ nie znamy jego adresu. Dlatego musimy przejść do elementu nr 1, aby sprawdzić adres elementu nr 2. Następnie w drugim elemencie znajdujemy adres trzeciego elementu itd., aż dojdziemy do ostatniego elementu. Listy powiązane są świetnym rozwiązaniem, gdy wiadomo, że i tak będzie trzeba odczytywać wszystkie elementy na raz: wówczas można odczytać jeden element, wziąć adres następnego, odczytać następny element, wziąć adres następnego itd. Jeżeli jednak zamierzamy odczytywać elementy nieregularnie, lista powiązana będzie bardzo złym wyborem.

Tablice są inne, ponieważ znany jest adres każdego elementu. Powiedzmy np., że nasza tablica zawiera pięć elementów i wiemy, iż adres pierwszego z nich to 00. Jaki jest zatem adres piątego elementu?



Nietrudno policzyć, że piąty element znajduje się pod adresem 04. Tablice są doskonałym wyborem, gdy elementy mają być odczytywane nie po kolejach, ponieważ dostęp do każdego z nich jest natychmiastowy. W liście powiązanej elementy nie znajdują się obok siebie, przez co nie można błyskawicznie obliczyć pozycji piątego z nich — trzeba przejść do pierwszego, sprawdzić adres drugiego, następnie przejść do drugiego, aby sprawdzić adres trzeciego itd., aż do osiągnięcia piątego elementu.

Terminologia

Elementy w tablicach są numerowane. Numeracja zwykle zaczyna się od 0, nie od 1. W poniżej tablicy np. wartość 20 znajduje się na pozycji nr 1.

10	20	30	40
0	1	2	3

Wartość 10 natomiast znajduje się na pozycji 0. Początkujący programiści często mają z tego powodu kłopoty. Rozpoczynanie numeracji od zera ułatwia wykonywanie wielu operacji na tablicach i dlatego programiści trzymają się tej metody. Jest ona stosowana w prawie wszystkich językach programowania i wkrótce na pewno się do niej przyzwyczaisz.

Pozycja elementu nazywa się **indeksem**. Dlatego zamiast mówić: „Wartość 20 znajduje się na pozycji 1” powinno się mówić: „Wartość 20 ma indeks 1”. Dalej w książce słowo *indeks* będzie oznaczać właśnie *pozycję*.

Oto czasy wykonywania operacji najczęściej stosowanych na tablicach i listach.

	Tablice	Listy
Odczyt	$O(1)$	$O(n)$
Wstawianie	$O(n)$	$O(1)$

$$O(n) = \text{czas liniowy}$$

$$O(1) = \text{czas stały}$$

Dlaczego czas wstawiania elementu do tablicy wynosi $O(n)$? Powiedzmy, że chcesz wstawić element na początku tablicy. Jak zrobiłbyś to? Ile czasu by to zajęło? Odpowiedzi na te pytania znajdziesz w następnych punktach!

ĆWICZENIE

- 2.1.** Wyobraź sobie, że tworzysz aplikację do zarządzania domowymi finansami.

1. warzywa
2. film
3. członkostwo w sfbc

Codziennie zapisujesz wszystkie swoje wydatki. Na koniec miesiąca przeglądzasz notatki i podsumowujesz, ile wydałeś. W związku z tym często dodajesz nowe elementy i rzadko odczytyujesz dane. Czy w takim razie dla Ciebie lepsza będzie tablica, czy lista?

Wstawianie elementów w środku listy

Powiedzmy, że chcesz, aby Twoja lista zadań do wykonania działała podobnie do kalendarza. Wcześniej dodawaliśmy elementy na końcu.

Teraz chcemy dodawać elementy w takiej kolejności, w jakiej powinny być realizowane zadania.

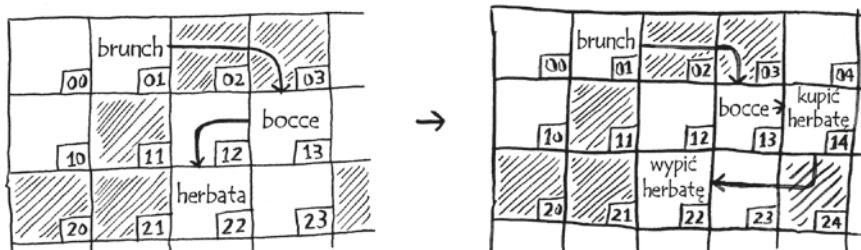


Nieuporządkowane

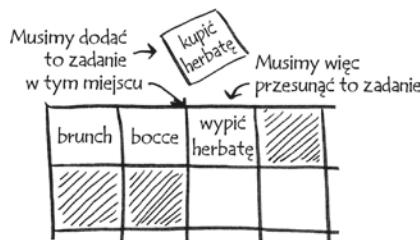


Uporządkowane

Która ze struktur lepiej się sprawdza, gdy trzeba wstawiąć elementy w środku, tablica czy lista? W liście wystarczy tylko zmienić wskaźnik w poprzednim elemencie i gotowe.



Natomiast w tablicy trzeba przesunąć o jedną pozycję wszystkie elementy znajdujące się za miejscem wstawienia.



Jeśli nie będzie miejsca na przesunięcie elementów, może być konieczne skopiowanie całej tablicy w nowe miejsce! Kiedy trzeba wstawiąć elementy w środku struktury, listy z całą pewnością są lepszym rozwiązaniem.

Usuwanie elementów

A gdybyśmy chcieli usunąć jakiś element? W takim przypadku listy również mają przewagę, ponieważ wystarczy tylko zmienić wskaźnik w poprzednim elemencie. W tablicy po usunięciu elementu trzeba by przesunąć wszystkie, które znajdowały się za nim.

Usunięcie, w odróżnieniu od wstawiania, zawsze się udaje. Elementu czasami nie udaje się wstawić z powodu braku miejsca w pamięci, natomiast usunąć element można zawsze.

Oto czasy wykonywania typowych operacji na tablicach i listach powiązanych.

	Tablice	Listy
Odczyt	$O(1)$	$O(n)$
Wstawianie	$O(n)$	$O(1)$
Usuwanie	$O(n)$	$O(1)$

Warto podkreślić, że czas operacji wstawiania i usuwania wynosi $O(1)$ tylko wtedy, gdy element do usunięcia jest natychmiast dostępny. Często zapisuje się pierwszy i ostatni element listy powiązanej, aby operacja ich usuwania miała czas wykonywania na poziomie $O(1)$.

Czego używa się częściej: tablic czy list powiązanych? Oczywiście wszystko zależy od potrzeb, ale tablice znajdują szerokie uznanie ze względu na szybki dostęp swobodny do elementów. Wyróżnia się dwa rodzaje dostępu do elementów, czyli **swobodny** i **sekwencyjny**. Ten drugi polega na tym, że elementy odczytuje się po kolej, zaczynając od pierwszego. Elementy listy powiązanej można przeglądać *tylko sekwencyjnie*. Jeśli trzeba odczytać dziesiąty element takiej listy, należy najpierw odczytać dziewięć poprzednich, aby po połączeniach dojść do dziesiątego. Z kolei dostęp swobodny umożliwia przejście bezpośrednio do dziesiątego elementu. Wielokrotnie będę podkreślał, że tablice zapewniają szybszy odczyt. Wynika to z tego, że struktury te charakteryzują się dostępem swobodnym. Taki sposób pobierania elementów znajduje zastosowanie w wielu przypadkach i dlatego tablice są bardzo często używane. Poza tym przy użyciu tablic i list powiązanych implementuje się też inne struktury danych, o których piszę dalej w tej książce.

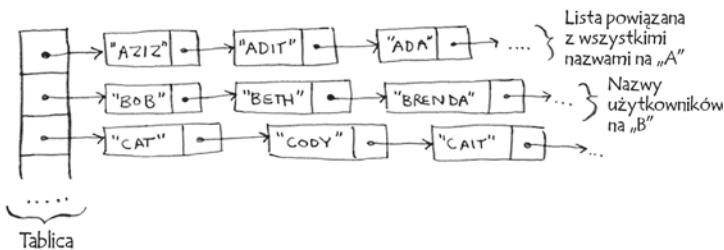
ĆWICZENIA

- 2.2.** Wyobraź sobie, że tworzysz aplikację do przyjmowania zamówień od klientów w restauracji. Jedną z funkcji jest zapisywanie listy zamówień. Kelnerzy cały czas dodają nowe zamówienia, a szefowie kuchni je pobierają i przygotowują potrawy. Jest to kolejka zamówień: kelnerzy dodają zamówienia na końcu kolejki, a szefowie kuchni pobierają je z początku kolejki i gotują dania.



Czy do implementacji takiej kolejki użyłbyś tablicy, czy listy powiązanej. (Podpowiedź: listy powiązane są dobre we wstawianiu i usuwaniu elementów, a tablice są lepsze w dostępie swobodnym. Które operacje będziesz wykonywać w tym przypadku?).

- 2.3.** Proponuję eksperyment myślowy. Powiedzmy, że Facebook przechowuje listę nazw użytkowników. Gdy ktoś próbuje zalogować się na konto, system szuka podanej przez tego kogoś nazwy użytkownika. Jeśli ją znajdzie, może nastąpić zalogowanie. Ludzie bardzo często logują się na Facebook, więc lista nazw użytkowników jest bardzo często przeszukiwana. Powiedzmy, że do jej przeszukiwania portal używa algorytmu wyszukiwania binarnego. Algorytm ten wymaga dostępu swobodnego — musi mieć natychmiastowy dostęp do środkowego elementu listy. Czy dysponując tą wiedzą, zaimplementowałbyś listę nazw użytkowników na bazie tablicy, czy listy powiązanej?
- 2.4.** Na Facebooku także bardzo często tworzone są nowe konta. Powiedzmy, że listę użytkowników postanowiliśmy przechowywać w tablicy. Jakie są wady tablicy, jeśli chodzi o wstawianie elementów? Innymi słowy, pomyśl sobie, że szukasz loginów za pomocą wyszukiwania binarnego. Co się stanie, gdy dodasz nowych użytkowników do tablicy?
- 2.5.** W rzeczywistości Facebook nie przechowuje informacji ani w tablicy, ani w liście powiązanej. Zastanówmy się więc nad hybrydową strukturą danych, czyli tablicą list powiązanych. Mamy tablicę z 26 miejscami. Każde z nich wskazuje listę powiązaną. Pierwsze miejsce np. wskazuje listę powiązaną zawierającą wszystkie nazwy użytkowników zaczynające się na literę *a*. Drugie miejsce wskazuje listę powiązaną zawierającą wszystkie nazwy użytkowników zaczynające się na literę *b* itd.



Powiedzmy, że na Facebooku postanawia założyć konto Adit B i chcemy dodać go do listy. Przechodzimy do pierwszego miejsca w tablicy, następnie idziemy do wskazywanej przez niego listy powiązanej i dodajemy Adit B na końcu. A teraz wyobraź sobie, że chcesz znaleźć użytkownika o nazwie Zakhir H. Idziemy więc do miejsca 26. w tablicy, które wskazuje listę powiązaną zawierającą wszystkie nazwy na Z. Następnie w liście tej szukamy użytkownika Zakhir H.

Porównaj tę strukturę danych z tablicami i listami powiązanymi. Czy będzie szybsza, czy wolniejsza podczas wyszukiwania i wstawiania danych w porównaniu z tymi strukturami? Nie musisz podawać czasów wykonywania w notacji dużego O — wystarczy, że powiesz, czy ta nowa struktura będzie szybsza, czy wolniejsza.

Sortowanie przez wybieranie

Poskładamy wszystko i wyjdzie nam nowy algorytm, czyli sortowanie przez wybieranie. Aby zrozumieć treść tego podrozdziału, należy znać tablice i listy oraz notację dużego O.

Powiedzmy, że masz w swoim komputerze trochę plików muzycznych i prowadzisz rejestr liczby odtworzeń utworów każdego artysty.

Chcesz posortować tę listę od najmniejszej liczby odtworzeń, aby utworzyć ranking swoich ulubionych wykonawców. Jak to zrobić?



~ ♫ ~	Liczba odtworzeń
Radiohead	156
Kishore Kumar	141
The Black Keys	35
Neutral Milk Hotel	94
Beck	88
The Strokes	61
Wilco	111

Jednym ze sposobów jest przejrzenie całej listy oraz znalezienie artysty, którego utwory są najczęściej odtwarzane, i dodanie go do nowej listy.

1. najczęściej odtwarzane są utwory Radiohead

~ ♫ ~	Liczba odtworzeń
Radiohead	156
Kishore Kumar	141
The Black Keys	35
Neutral Milk Hotel	94
Beck	88
The Strokes	61
Wilco	111

2. dodanie pozycji do nowej listy

♪ Posortowane ♪	Liczba odtworzeń
Radiohead	156

Następnie w taki sam sposób szukamy drugiego najpopularniejszego wykonawcy.

1. na drugim miejscu znajduje się kishore kumar

~ ♫ ~	Liczba odtworzeń
Kishore Kumar	141
The Black Keys	35
Neutral Milk Hotel	94
Beck	88
The Strokes	61
Wilco	111

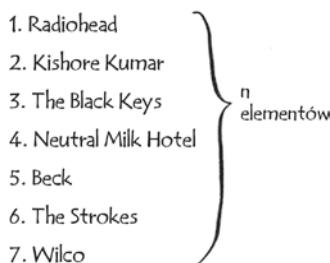
2. dodajemy więc tego wykonawcę na drugim miejscu w nowej liście

♪ Posortowane ♪	Liczba odtworzeń
Radiohead	156
Kishore Kumar	141

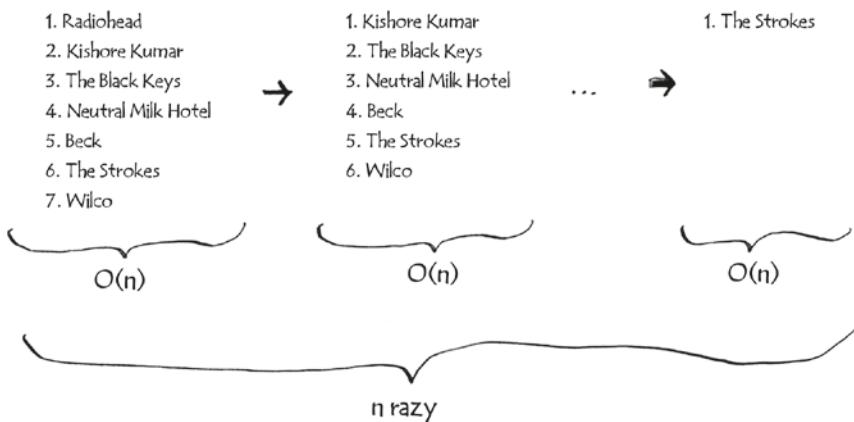
Kontynuując ten sposób działania, otrzymujemy posortowaną listę.

~ ♫ ~	Liczba odtworzeń
Radiohead	156
Kishore Kumar	141
Wilco	111
Neutral Milk Hotel	94
Beck	88
The Strokes	61
The Black Keys	35

Wcielimy się na chwilę w rolę naukowców zajmujących się algorytmiką i zastanowimy się, ile czasu zajmie takie sortowanie. Przypomnij, że $O(n)$ oznacza, że każdy element listy zostanie dotknietym raz. Przykładowo proste przeszukiwanie listy wykonawców wymaga obejrzenia każdego elementu tej listy tylko raz.



Aby znaleźć artystę, którego utwory były słuchane najczęściej, należy sprawdzić wszystkie elementy na liście. Jak już się przekonaliśmy, operacja taka zajmuje $O(n)$ czasu. Mamy więc operację o czasie wykonywania $O(n)$, którą musimy wykonać n razy.



Zatem czas wykonywania wynosi $O(n \cdot n)$ czyli $O(n^2)$.

Algorytmy sortowania mają wiele zastosowań. Za ich pomocą można uporządkować:

- nazwiska w książce telefonicznej,
- daty podróży,
- adresy e-mail (od najnowszych).

Malejąca liczba elementów do sprawdzenia

Niektórzy pewnie się zastanawiają, czy przypadkiem nie jest tak, że w miarę wykonywania operacji liczba elementów do przejrzenia maleje. W końcu pozostało już tylko jeden element do sprawdzenia. Skoro tak, to jakim cudem czas wykonywania wynosi $O(n^2)$? To dobre pytanie, a odpowiedź jest związana ze stałymi w notacji dużego O. Szerzej opisuję tę kwestię w rozdziale 4., a poniżej przedstawiam tylko krótkie wprowadzenie.

To prawda, że nie trzeba sprawdzać n elementów za każdym razem. Najpierw do sprawdzenia jest n elementów, potem $n - 1, n - 2 \dots 2, 1$. Średnio do sprawdzenia jest lista zawierająca $1/2 \cdot n$ elementów. Zatem czas wykonywania wynosi $O(n \cdot 1/2 \cdot n)$. Jednak w notacji dużego O ignoruje się stałe, takie jak $1/2$ (przypominam, że szczegółowe wyjaśnienie tego tematu zamieściłem w rozdziale 4.), i dlatego powinniśmy napisać $O(n \cdot n)$, czyli $O(n^2)$.

Sortowanie przez wybieranie to ciekawy, choć niezbyt szybki algorytm. Szybszy jest algorytm szybkiego sortowania (ang. *quicksort*), który charakteryzuje się czasem wykonywania $O(n \log n)$. Opisuję go w rozdziale 4.

Przykładowa implementacja

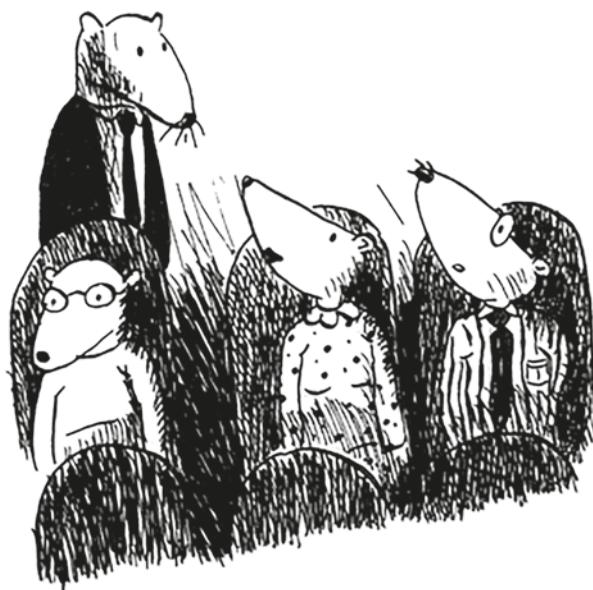
Nie pokazałem kodu sortującego listę artystów, ale poniżej przedstawiam program o podobnym zastosowaniu — sortujący tablicę w porządku rosnącym. Oto funkcja znajdująca najmniejszy element w tablicy.

```
def findSmallest(arr):
    smallest = arr[0]      ← ..... Przechowuje najmniejszą wartość.
    smallest_index = 0      ← ..... Przechowuje indeks najmniejszej wartości.
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index
```

Przy użyciu tej funkcji możemy napisać algorytm sortowania przez wybieranie.

```
def selectionSort(arr):   ← ..... Sortuje tablicę.
    newArr = []
    for i in range(len(arr)):
        smallest = findSmallest(arr) ← ..... Znajduje najmniejszy element
                                         w tablicy i dodaje go do nowej tablicy.
        newArr.append(arr.pop(smallest))
    return newArr

print selectionSort([5, 3, 6, 2, 10])
```



Powtórzenie

- Pamięć komputera jest jak gigantyczna szafa z szufladami.
- Do przechowywania wielu elementów należy używać tablic i list.
- Elementy tablicy są przechowywane obok siebie w sposób ciągły.
- Elementy listy mogą być porozrzucane po całej pamięci i każdy z nich zawiera adres następnego.
- Tablice zapewniają szybki odczyt elementów.
- Listy powiązane umożliwiają szybkie wstawianie i usuwanie elementów.
- Wszystkie elementy w tablicy powinny być tego samego typu (np. `int`, `double` itd.).



W tym rozdziale:

- poznasz rekurencję, czyli technikę stosowaną w implementacji wielu algorytmów. Musisz ją zrozumieć, aby przejść do następnych rozdziałów tej książki.
- dowiesz się, jak podzielić problem na przypadek podstawowy i przypadek rekurencyjny. Tę prostą koncepcję wykorzystuje się np. w algorytmach typu dziel i zwyciężaj (rozdział 4.) używanych do rozwiązywania trudnych zadań.

Jestem podekscytowany, ponieważ w tym rozdziale opisuję **rekurencję**, która jest bardzo elegancką metodą rozwiązywania zadań. Zaliczam ją do moich ulubionych tematów, ale naprawdę zdania na jej temat są podzielone. Niektórzy ludzie kochają rekurencję, a inni jej nienawidzą, choć są też tacy, którzy zakochują się w niej dopiero po kilku latach używania. Sam należę właśnie do tej trzeciej grupy. Aby było Ci łatwiej, poniżej zamieszczam kilka rad.

- W rozdziale tym przedstawiam wiele przykładów kodu. Uruchom każdy z nich, aby dokładnie sprawdzić, jak działa.
- Będę opisywał funkcje rekurencyjne. Przynajmniej raz prześledź krok po kroku wykonywanie takiej funkcji przy użyciu ołówka i kawałka papieru. Zrób to mniej więcej tak: „Niech zobaczę, co się stanie, gdy przekażę 5 do funkcji factorial, a potem zwrócę iloczyn 5 razy 4 do factorial, który wynosi...” itd. Takie analizowanie działania funkcji rekurencyjnych ułatwia ich zrozumienie.

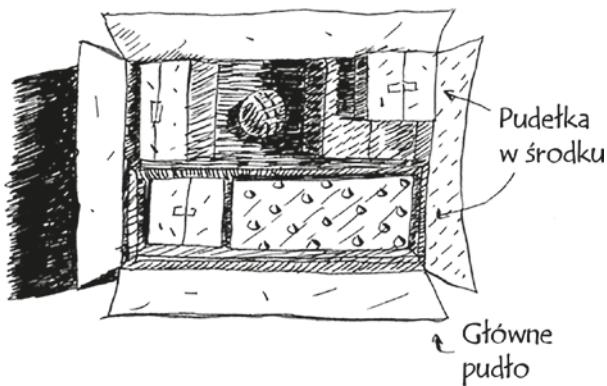
Ponadto w rozdziale tym znajduje się dużo pseudokodu. **Pseudokod** to wysokopoziomowy opis rozwiązywanego problemu. Wygląda jak prawdziwy kod źródłowy, ale jest bliższy językowi ludzkiemu.

Rekurencja

Wyobraź sobie, że buszując na strychu u babci, znajdujesz tajemniczą, zamkniętą na kłódkę walizkę.



Babcia mówi, że klucz powinien być gdzieś w tym pudełku.



Pudło to zawiera inne pudełka, w których znajdują się kolejne, jeszcze mniejsze pudełka. Klucz jest w jednym z nich. Jakiego algorytmu użyjesz, aby go znaleźć? Dobrze się nad tym zastanów, zanim przeczytasz następne akapity.

Oto propozycja rozwiązania.



1. Ułóż pudełka w stos do przejrzenia.
2. Weź pudełko i do niego zajrzyj.
3. Jeśli znajdziesz pudełko, dodaj je do stosu, aby zatrzymać go później.
4. Jeśli znajdziesz klucz, możesz zakończyć poszukiwania!
5. Powtórz czynności.

A to inna możliwość.



1. Przejrzyj zawartość pudełka.
2. Jeśli znajdziesz pudełko, wróć do punktu 1.
3. Jeśli znajdziesz klucz, kończysz poszukiwania!

Która metoda wydaje Ci się łatwiejsza? W pierwszym przypadku korzystamy z pętli `while`. Dopóki stos nie jest pusty, pobieramy z niego po jednym pudełku i przeglądamy jego zawartość.

```
def look_for_key(main_box):
    pile = main_box.make_a_pile_to_look_through()
    while pile is not empty:
        box = pile.grab_a_box()
        for item in box:
            if item.is_a_box():
                pile.append(item)
            elif item.is_a_key():
                print "Znaleziono klucz!"
```

W drugim rozwiążaniu wykorzystano rekurencję. **Rekurencja** występuje wtedy, gdy funkcja wywołuje samą siebie. Oto pseudokod przedstawiający możliwy sposób realizacji tego rozwiązania.

```
def look_for_key(box):
    for item in box:
        if item.is_a_box():
            look_for_key(item) ←..... Rekurencja!
        elif item.is_a_key():
            print "Znaleziono klucz!"
```

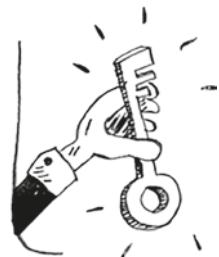
Obie metody prowadzą do tego samego celu, ale uważam, że druga jest mniej skomplikowana. Właśnie w tych przypadkach używa się rekurencji, gdy pozwala uprościć rozwiązanie problemu. Z racji jej zastosowania nie wynika żadna korzyść pod względem szybkości działania programu. Czasami pętle są nawet szybsze. Bardzo podoba mi się wypowiedź Leigha Caldwella na Stack Overflow: „Pętle mogą przyspieszyć program. Rekurencja może przyspieszyć programistę. Zdecyduj, co jest ważniejsze w danej sytuacji!”¹.

Rekurencja jest wykorzystywana w wielu ważnych algorytmach i dlatego tak ważne jest zrozumienie tej koncepcji.

Przypadki podstawowy i rekurencyjny

Ponieważ funkcja rekurencyjna wywołuje samą siebie, nietrudno popełnić błąd i utworzyć nieskończoną pętlę. Powiedzmy np., że chcemy napisać funkcję odliczania w dół, taką jak poniższa.

1. <http://stackoverflow.com/a/72694/139117>



> 3...2...1

Przy zastosowaniu rekurencji funkcja ta mogłaby wyglądać tak.

```
def countdown(i):
    print i
    countdown(i-1)
```

Przepisz ten kod i go uruchom. Od razu dostrzeżesz problem: ta funkcja nigdy się nie kończy!



Nieskończona pętla

> 3...2...1...0...-1...-2...

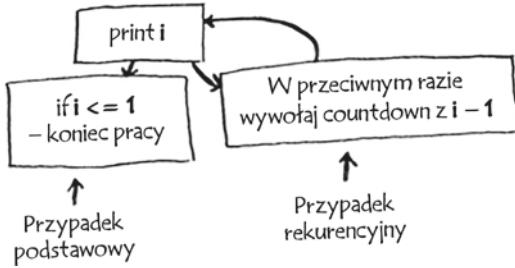
(Naciśnij klawisze *Ctrl+C*, aby zakończyć skrypt).

Pisząc funkcję rekurencyjną, należy zadbać o warunek zakończenia jej działania. Dlatego właśnie *każda funkcja rekurencyjna składa się z dwóch części, czyli z przypadku podstawowego i przypadku rekurencyjnego*. Przypadek rekurencyjny zachodzi wtedy, gdy funkcja wywołuje samą siebie. Natomiast przypadek podstawowy zachodzi wtedy, gdy funkcja nie wywołuje ponownie samej siebie..., a więc nie wpada w pętlę nieskończoną.

Do naszej funkcji odliczającej dodamy przypadek podstawowy.

```
def countdown(i):
    print i
    if i <= 0:   ← ..... Przypadek podstawowy.
        return
    else:   ← ..... Przypadek rekurencyjny.
        countdown(i-1)
```

Teraz funkcja działa, tak jak powinna, co można zobaczyć na poniższym rysunku.



Stos

W tym podrozdziale opisuję **stos wywołań**. Pojęcie to jest bardzo ważne w programowaniu w ogóle, ale musi je rozumieć także każdy, kto chce nauczyć się używania rekurencji.



Powiedzmy, że organizujesz grill. Napisałś sobie listę czynności do wykonania w postaci stosu przyklejanych karteczek.



Pamiętasz z części poświęconej tablicom i listom listę czynności do wykonania? Na liście elementy można swobodnie dodawać i usuwać. Stos przyklejanych karteczek jest o wiele prostszy. Każdy nowy element jest dodawany na początku listy. Także każda operacja odczytu zwraca pierwszy element, który następnie zostaje usunięty z listy. W związku z tym nasza lista czynności do wykonania ma tylko dwie operacje; są to **push** (wstawianie) i **pop** (usuwanie i odczyt).



push
(dodanie nowego
elementu na początek)

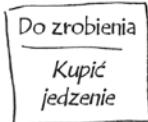


pop
(odczyt i usunięcie
pierwszego elementu)

Zobaczmy, jak taka lista działa w praktyce.



Pobranie
zadania ze stosu



Na kartce jest napisane
„Kupić jedzenie”. Trzeba kupić
bułki, burgery i upiec ciasto



Wstawmy te
karteczki na stos

Taką strukturę danych nazywamy **stosem**. Jest to bardzo prosta struktura. Korzystałeś z niej przez cały ten czas, nawet o tym nie wiedząc!

Stos wywołań

Twój komputer w swoich wewnętrznych operacjach wykorzystuje stos zwany **stosem wywołań**. Zobaczmy go w akcji. Oto prosta funkcja.

```
def greet(name):  
    print "Cześć, " + name + "!"  
    greet2(name)  
    print "Przygotowywanie do pożegnania..."  
    bye()
```

Funkcja ta witą się z użytkownikiem, a następnie wywołuje dwie inne funkcje. Oto one.

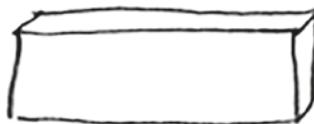
```
def greet2(name):  
    print "Jak się masz, " + name + "?"  
  
def bye():  
    print "OK, cześć!"
```

Przeanalizujmy dokładnie, co się dzieje, gdy wywoływana jest funkcja.

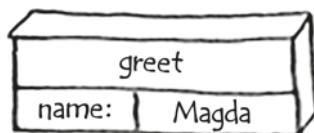
Uwaga

Funkcja `print` jest składnikiem języka Python, ale dla ułatwienia będziemy udawać, że nie jest. Po prostu zaufaj mi.

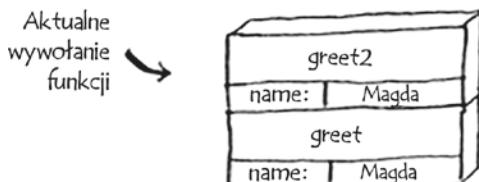
Powiedzmy, że wywołujemy funkcję `greet("Magda")`. Najpierw komputer przydzieli dla tego wywołania funkcji blok pamięci.



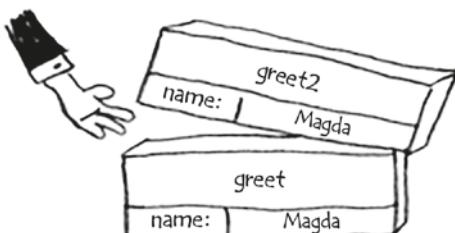
Teraz możemy posłużyć się tym blokiem. Zmiennej o nazwie `name` zostaje przypisana wartość "Magda". Musimy ją zapisać w pamięci.



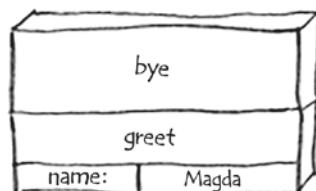
Za każdym razem, gdy wywołujesz funkcję, komputer zapisuje wartości wszystkich zmiennych dotyczących tego wywołania w pamięci. Następnie drukujemy napis: Cześć, Magda!. Później wywołujemy `greet2("Magda")`. I znowu komputer przydziela blok pamięci dla wywołania funkcji.



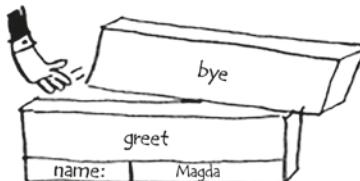
Komputer przechowuje te pudełka na stosie. Drugie pudełko zostaje postawione na pierwszym. Program drukuje napis: Jak się masz Magda?, a później następuje zdjęcie pudełka z wierzchu stosu.



Teraz na wierzchu stosu znajduje się pudełko funkcji `greet`, co znaczy, że wróciliśmy właśnie do niej. Gdy wywołaliśmy funkcję `greet2`, funkcja `greet` była częściowo wykonana. To jest najważniejsza informacja w tym podrozdziale: *gdy wywoływana jest funkcja wewnętrz innej funkcji, to funkcja nadzędna zostaje wstrzymana w stanie częściowego wykonania*. Wszystkie wartości zmiennych tej funkcji nadzędnej nadal znajdują się w pamięci. Po zakończeniu pracy z funkcją `greet2` wracamy do funkcji `greet` i wznowiamy jej wykonywanie od miejsca, w którym skończyliśmy. Najpierw drukujemy napis: Przygotowywanie do pożegnania..., a następnie wywołujemy funkcję `bye`.



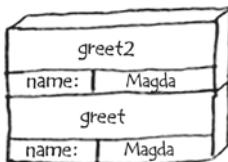
Do stosu zostaje dodane pudełko reprezentujące funkcję `bye`, następuje wydrukowanie napisu: `OK, cześć!` i znów wracamy do dalszego wykonywania funkcji nadrzędnej.



Z powrotem jesteśmy w funkcji `greet`. Nie ma już więcej instrukcji do wykonania, więc z tej funkcji też wychodzimy. Opisany stos, służący do przechowywania wartości zmiennych różnych funkcji, nazywa się **stosem wywołań**.

ĆWICZENIE

3.1. Spójrz na poniższy stos wywołań.



Jakie informacje możesz podać na podstawie tego stosu?

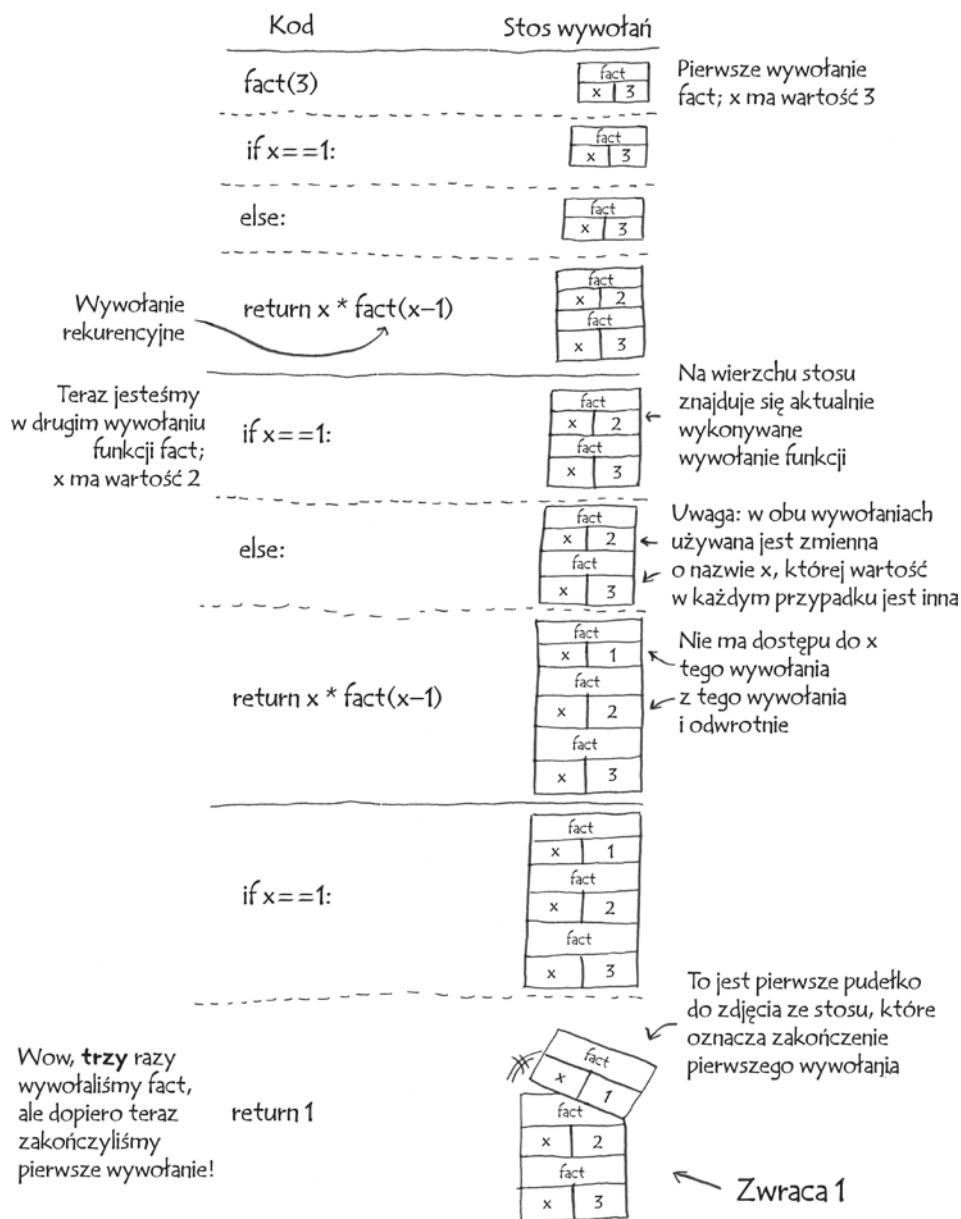
Teraz zobaczymy, co się dzieje na stosie wywołań podczas wykonywania funkcji rekurencyjnej.

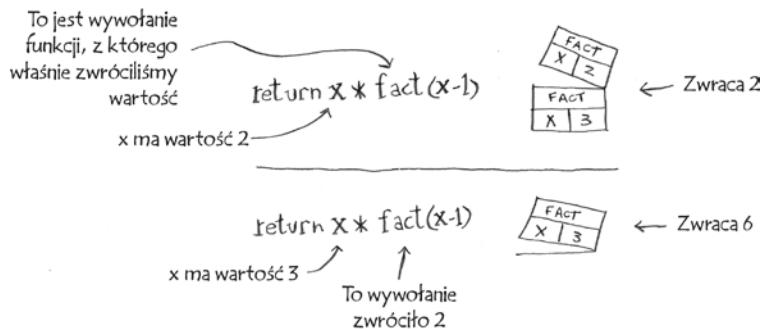
Stos wywołań z rekurencją

Funkcje rekurencyjne także korzystają ze stosu! Przyjrzymy się, jak to robią, na przykładzie funkcji `factorial`. Wywołanie `factorial(5)` to odpowiednik matematycznej formuły $5!$, której rozwinięcie to: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$. Analogicznie `factorial(3)` to $3 \cdot 2 \cdot 1$. Poniżej znajduje się definicja funkcji rekurencyjnej obliczającej silnię podanej liczby.

```
def fact(x):
    if x == 1:
        return 1
    else:
        return x * fact(x-1)
```

Weźmy np. wywołanie funkcji `fact(3)`. Przeanalizujemy je krok po kroku, aby dokładnie prześledzić, co będzie działało się na stosie. Przypomnę, że pierwszy element na stosie odpowiada aktualnie wykonywanemu wywołaniu funkcji `fact`.



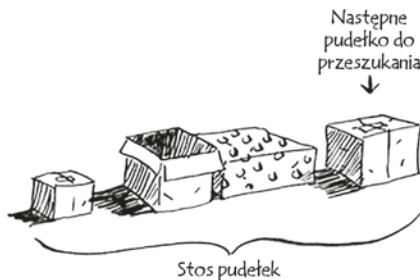


Zauważ, że każde wywołanie funkcji `fact` ma własny egzemplarz zmiennej `x`, do którego nie ma dostępu z innych wywołań tej funkcji.

Stos jest w rekurencji bardzo ważnym czynnikiem. W pierwszym przykładzie opisałem dwie możliwości znalezienia klucza. Oto przypomnienie pierwszej z nich.



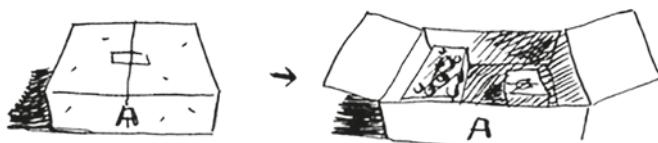
W ten sposób budujemy stos pudełek do przeszukania, dzięki czemu zawsze wiemy, które pudełka trzeba jeszcze sprawdzić.



Natomiast w metodzie rekurencyjnej nie ma stosu.

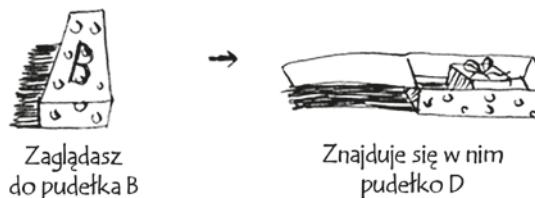


Jeśli nie ma żadnego stosu, to skąd algorytm „ma wiedzieć”, które pudełka pozostały jeszcze do obejrzenia? Oto przykład.



Zaglądasz do pudełka A

W środku są pudełka B i C



Zaglądasz do pudełka B

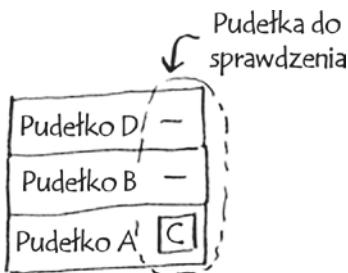
Znajduje się w nim pudełko D



Zaglądasz do pudełka D

Jest puste

W tym momencie stos wygląda tak.



„Stos pudełek” jest przechowywany na stosie! Zawiera on częściowo wykonyane wywołania funkcji, z których każde ma własną częściową listę pudełek do sprawdzenia. Stos to bardzo wygodne rozwiązanie, ponieważ uwalnia od konieczności samodzielnie pilnowania kolejności przeglądania pudełek.

Jednak wygoda, jaką zapewnia stos, nie jest darmowa, gdyż do zapisania wszystkich tych informacji potrzeba dużo pamięci. Każde wywołanie funkcji zajmuje trochę miejsca i gdy stos jest bardzo wysoki, wiadomo, że komputer zapisał już dane wielu wywołań funkcji. W takim przypadku są dwie możliwości do wyboru:

- zamienić rekurencję na pętle,
- skorzystać z tzw. **rekurencji ogonowej** (ang. *tail recursion*). Jest to zaawansowana technika, na opis której nie ma miejsca w tej książce. Ponadto obsługują ją tylko niektóre języki programowania.

ĆWICZENIE

3.2. Wyobraź sobie, że przez przypadek napisałś nieskończoną funkcję rekurencyjną. Jak już wiesz, dla każdego wywołania funkcji komputer przydziela pamięć na stosie. Co się dzieje ze stosem, gdy funkcja nigdy się nie kończy?

Powtórzenie

- Rekurencja to technika polegająca na tym, że funkcja wywołuje samą siebie.
- Każda funkcja rekurencyjna ma dwa przypadki: podstawowy i rekurencyjny.
- Na stosie można wykonać dwie operacje: push (wstawianie elementów) i pop (zdejmowanie elementów).
- Wszystkie wywołania funkcji trafiają na stos wywołań.
- Stos wywołań może być bardzo duży i zajmować dużo pamięci.





W tym rozdziale:

- poznasz technikę o nazwie „dziel i rządź”. Z pewnością nieraz natkniesz się na problem, którego nie będzie się dało rozwiązać przy użyciu żadnego znanego Ci algorytmu. Dobry specjalista, napotykając takie wyzwanie, nigdy się nie poddaje. Ma w zanadrzu wiele technik, które może zastosować, aby jakoś rozwiązać problem. Pierwszą taką ogólną techniką, jakiej się nauczysz, jest właśnie metoda „dziel i rządź”.
- poznasz algorytm szybkiego sortowania, który dzięki swojej elegancji często znajduje praktyczne zastosowanie. W algorytmie tym wykorzystywana jest metoda „dziel i rządź”.

W poprzednim rozdziale dokładnie wyjaśniłem, na czym polega rekurencja. W tym rozdziale natomiast pokażę, jak wykorzystać nowo nabycie umiejętności do rozwiązywania prawdziwych problemów. Nauczysz się posługiwania rekurencyjną techniką „**dziel i rządź**” (ang. *divide and conquer*), która jest powszechnie wykorzystywana do rozwiązywania rozmaitych zadań.

W tym rozdziale nareszcie dochodzimy do sedna algorytmiki. W końcu algorytm rozwiązujący tylko jeden rodzaj problemów nie byłby zbyt przydatny. Technika „dziel i rządź” rzuca całkiem nowe światło na kwestię rozwiązywania problemów. Jest to kolejne narzędzie w Twoim zestawie. Gdy spotkasz nowy

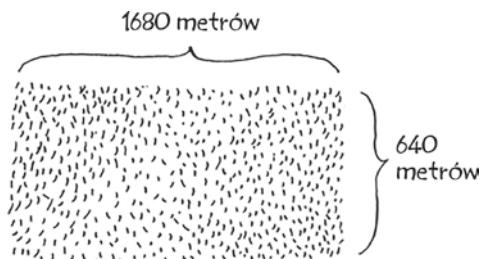
problem, wiesz, że to jeszcze nie koniec. Możesz się zastanowić: „Czy da się to rozwiązać za pomocą techniki «dziel i rządź»?”.

Dalej w tym rozdziale poznasz pierwszy poważny algorytm typu „dziel i rządź”, czyli **sortowanie szybkie** (ang. *quicksort*). Jest to oczywiście algorytm sortowania, który charakteryzuje się znacznie większą prędkością niż sortowanie przez wybieranie (opisane w rozdziale 2.). Ponadto jego implementacja może być doskonałą ilustracją eleganckiego kodu.

„Dziel i rządź”

Zrozumienie metody „dziel i rządź” może wymagać czasu. Dlatego opisuję ją na podstawie aż trzech przykładów. Najpierw przedstawiam przykład wizualny, potem przykład kodu źródłowego, który może nie jest superelegancki, ale za to łatwy do zrozumienia, a na koniec pokazuję algorytm sortowania szybkiego.

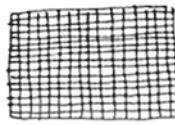
Powiedzmy, że jesteś rolnikiem posiadającym kawałek ziemi.



Chcesz podzielić tę działkę (będącą Twoim gospodarstwem) na równe **kwadraty**. Chcesz, aby kwadraty były jak największe, więc poniższe rozwiązania odpadają.



Pola nie są kwadratowe



Pola są za małe



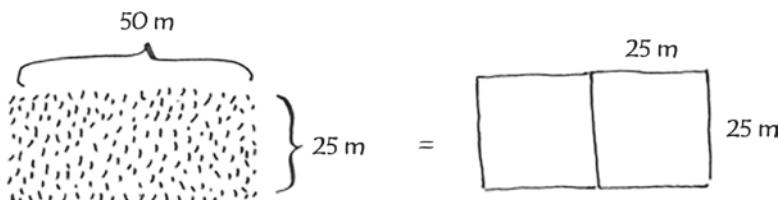
Wszystkie pola muszą być tego samego rozmiaru

Jak znaleźć największy możliwy rozmiar kwadratu do podzielenia kawałka ziemi na równe części? Najlepiej skorzystać z techniki „dziel i rządź”! Algorytmy z tej rodziny należą do algorytmów rekurencyjnych. Rozwiążanie problemu metodą „dziel i rządź” składa się z dwóch części. Oto one.

1. Określenie przypadku podstawowego, który powinien być najprostszym możliwym przypadkiem.
2. Dzielenie lub redukowanie problemu aż do sprowadzenia go do przypadku podstawowego.

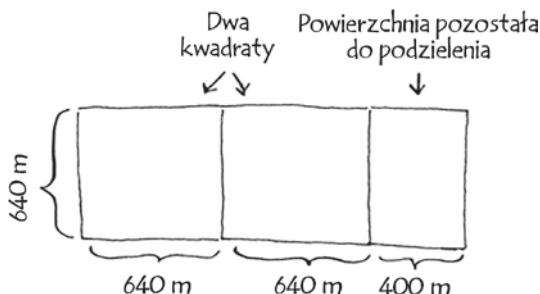
Poszukamy rozwiązania naszego problemu za pomocą techniki „dziel i rządź”. Jaki jest maksymalny rozmiar kwadratu, którego możemy użyć?

Najpierw szukamy przypadku podstawowego. Najprościej byłoby, gdyby jedna strona była wielokrotnością drugiej.

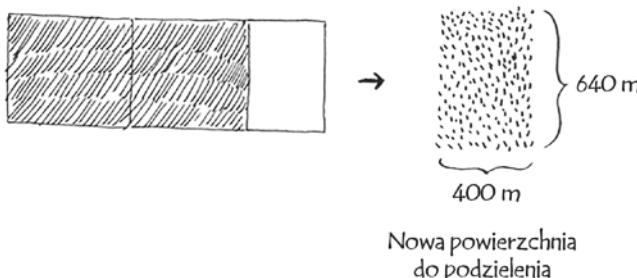


Powiedzmy, że jeden bok ma 25, a drugi — 50 metrów długości. Wówczas największy możliwy kwadrat miałby bok 25 m i wystarczyłyby dwa takie kwadraty, aby podzielić całą działkę.

Teraz musimy określić przypadek rekurencyjny. Do tego przyda się metoda „dziel i rządź”. Zgodnie z tą techniką każde rekurencyjne wywołanie musi skutkować redukcją problemu. Jak zredukować problem w tym zadaniu? Zaczniemy od zaznaczenia największych kwadratów, jakich moglibyśmy użyć.



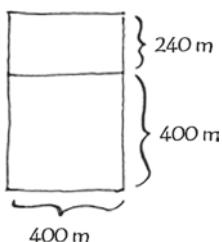
W obszarze do podzielenia zmieszcza się dwa kwadraty o boku 640 m i zostańcie jeszcze trochę wolnego miejsca. Teraz następuje olśnienie! Został jeszcze kawałek ziemi do podzielenia. *Do niego też można zastosować ten sam algorytm!*



Zatem początkowo mieliśmy do podzielenia obszar o wymiarach 1680×640 metrów, a teraz mamy już mniejszy kawałek o wymiarach 640×400 metrów. *Jeśli znajdziemy największy możliwy kwadrat dla tego obszaru, będzie to także największy możliwy kwadrat dla całego gospodarstwa.* W ten sposób zredukowaliśmy problem z wymiarów 1680×640 do 640×400 metrów!

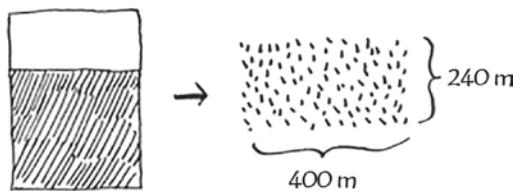
Algorytm Euklidesa

„Jeśli znajdziemy największy możliwy kwadrat dla tego obszaru, będzie to także największy możliwy kwadrat dla całego gospodarstwa”. Nie przejmuj się, jeśli to stwierdzenie nie jest dla Ciebie oczywiste, ponieważ nie jest oczywiste. Niestety dowód jego prawdziwości jest trochę za długi, aby przedstawić go w tej książce, więc musisz mi uwierzyć na słowo. Jeśli jednak chcesz to dokładnie zrozumieć, poszukaj informacji o algorytmie Euklidesa. Dobre objaśnienie znajduje się np. w Khan Academy pod adresem <https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm>.

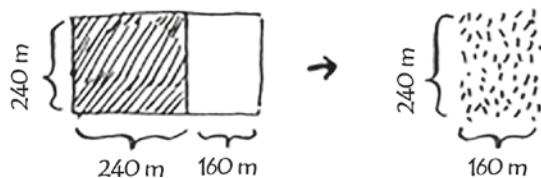


Ponownie skorzystamy z tego samego algorytmu. Jeśli gospodarstwo ma wymiary 640×400 metrów, to największy kwadrat, jaki można w nim utworzyć, ma bok 400 metrów.

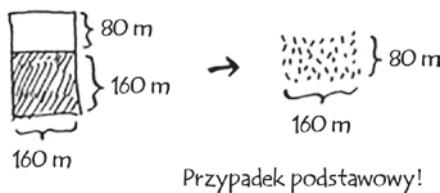
W ten sposób otrzymujemy kolejny zmniejszony skrawek o wymiarach 400×240 metrów.



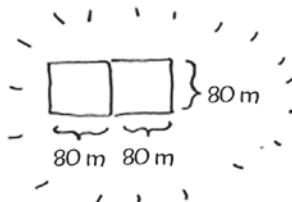
W nim możemy narysować kwadrat pozwalający otrzymać jeszcze mniejszy obszar o wymiarach 240×160 metrów.



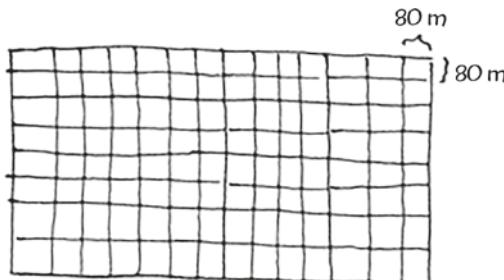
Następnie rysujemy kolejny kwadrat, aby otrzymać jeszcze mniejszy fragment.



Hej, dotarliśmy do przypadku bazowego, gdyż 80 jest czynnikiem liczby 160. Jeśli podzielimy ten obszar na dwa kwadraty, nic więcej nam nie zostanie!



Zatem największy możliwy rozmiar pola na naszej działce wynosi 80×80 metrów.



A to przypomnienie, jak działa technika „dziel i rządź”.

1. Określenie prostego przypadku podstawowego.
2. Redukowanie problemu aż do sprowadzenia go do przypadku podstawowego.

„Dziel i rządź” nie jest prostym algorytmem, za pomocą którego można rozwiązywać problemy. Jest to sposób analizowania problemów. Przyjrzymy się jeszcze jednemu przykładowi.

2	4	6
---	---	---

Dana jest tablica liczb.

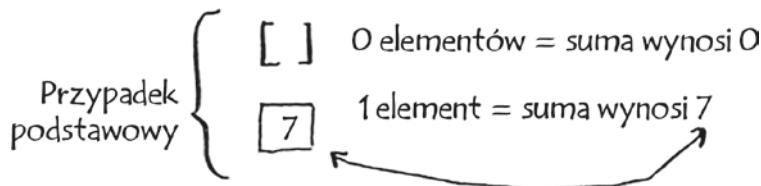
Naszym zadaniem jest zsumować wszystkie liczby i zwrócić sumę. Bardzo łatwo można to zrobić za pomocą pętli.

```
def sum(arr):
    total = 0
    for x in arr:
        total += x
    return total

print sum([1, 2, 3, 4])
```

Ale jak wykonać to zadanie przy użyciu funkcji rekurencyjnej?

Krok 1. Określenie przypadku podstawowego. Jaka jest najprostsza możliwa tablica? Pomyśl o najprostszym możliwym przypadku, a potem czytaj dalej. Jeśli tablica jest pusta lub zawiera jeden element, to zsumowanie jej będzie bardzo łatwe.



Krok 2. W każdym rekurencyjnym wywołaniu musimy zbliżać się do pustej tablicy. Jak zredukować rozmiar tego problemu? Oto jedna z możliwości.

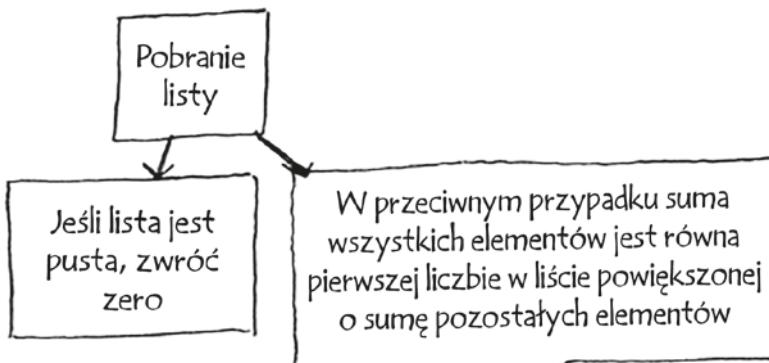
$$\text{sum} \left(\begin{array}{|c|c|c|} \hline 2 & 4 & 6 \\ \hline \end{array} \right) = 12$$

Wynik tego będzie taki sam jak tego.

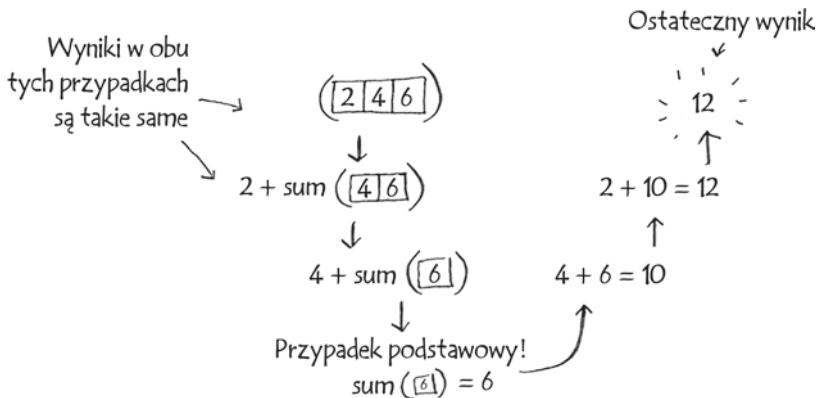
$$2 + \text{sum} \left(\begin{array}{|c|c|} \hline 4 & 6 \\ \hline \end{array} \right) = 2 + 10 = 12$$

W obu przypadkach suma wynosi 12, ale w drugiej wersji do funkcji `sum` przekazujemy mniejszą tablicę. Znaczy to, że zmniejszyliśmy rozmiar naszego problemu!

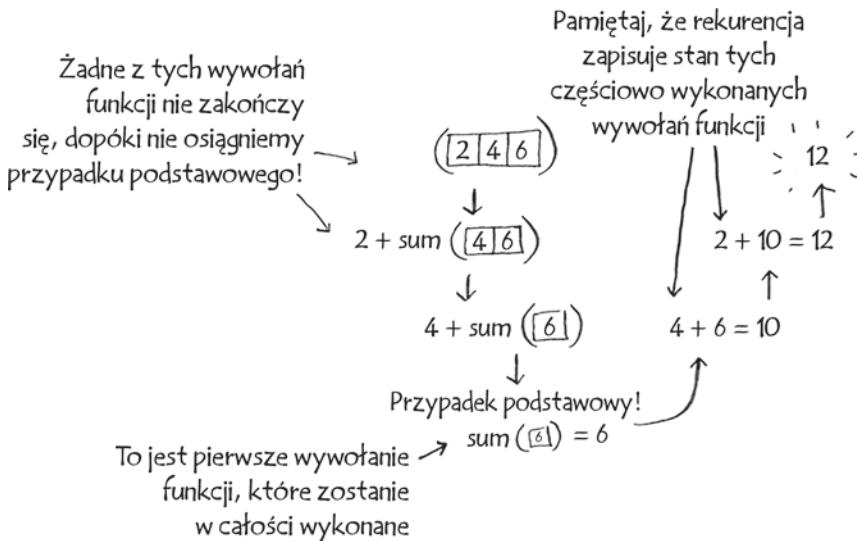
Nasza funkcja `sum` mogłaby działać w następujący sposób.



Oto praktyczna ilustracja.



Pamiętaj, że w rekurencji rejestrowany jest stan.



Wskazówka

Pisząc rekurencyjną funkcję operującą na tablicy, warto pamiętać, że przypadkiem podstawowym często może być tablica pusta lub zawierająca tylko jeden element. Jeśli nie wiesz, co zrobić, spróbuj najpierw tego rozwiązania.

Tajniki programowania funkcyjnego

Może się zastanawiasz: „Po co szukać rozwiązań rekurencyjnego, skoro bez problemu mogę sobie poradzić przy użyciu pętli”? Otóż w ten sposób niepostrzeżenie zostałeś wprowadzony w podstawowe tajniki programowania funkcyjnego! Funkcyjne języki programowania, takie jak Haskell, nie zawierają pętli, więc do pisania tego rodzaju funkcji, co opisana wyżej, należy używać rekurencji. Jeśli dobrze zrozumiesz rekurencję, łatwiej nauczysz się programowania funkcyjnego. Oto przykładowa implementacja funkcji `sum` w Haskellu.

```
sum [] = 0   ..... Przypadek podstawowy.  
sum (x:xs) = x + (sum xs) ..... Przypadek rekurencyjny.
```

Zauważ, że wygląda to tak, jakbyśmy napisali dwie definicje tej samej funkcji. Pierwsza zostanie wykonana w chwili dotarcia do przypadku podstawowego, a druga będzie wykonywana dla przypadków rekurencyjnych. Ewentualnie w Haskellu można napisać tę funkcję przy użyciu instrukcji `if`.

```
sum arr = if arr == []  
           then 0  
           else (head arr) + (sum (tail arr))
```

Jednak pierwsza definicja jest bardziej czytelna. Ponieważ w Haskellu rekurencja jest powszechnie wykorzystywana, język ten zawiera liczne udogodnienia, które ułatwiają jej stosowanie. Jeśli lubisz rekurencję albo chcesz nauczyć się nowego języka programowania, zachęcam do zainteresowania się Haskellem.

ĆWICZENIA

- 4.1.** Napisz kod źródłowy wcześniejszej funkcji `sum`.
- 4.2.** Napisz funkcję rekurencyjną liczącą elementy w liście.
- 4.3.** Znajdź największą liczbę w liście.
- 4.4.** Pamiętasz wyszukiwanie binarne z rozdziału 1.? To też jest algorytm typu „dziel i rządź”. Potrafisz określić przypadki bazowy i rekurencyjny dla wyszukiwania binarnego?



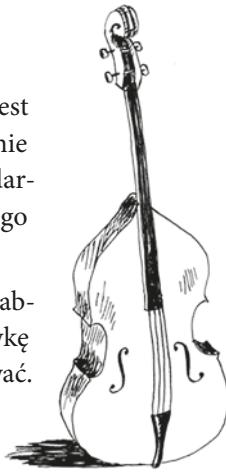
Sortowanie szybkie

Sortowanie szybkie (ang. *quicksort*) to oczywiście algorytm sortowania. Jest znacznie szybszy od sortowania przez wybieranie i dzięki temu powszechnie wykorzystywany w wielu prawdziwych programach. W bibliotece standardej języka C dostępna jest np. funkcja `qsort` będąca implementacją tego opartego na technice „dziel i rządź” algorytmu.

Użyjemy sortowania szybkiego do posortowania tablicy. Jaką najprostszą tablicę może obsłużyć algorytm sortowania (przypomnij sobie moją wskazówkę z poprzedniego podrozdziału)? Niektórych tablic w ogóle nie trzeba sortować.

Takich tablic
nie trzeba
sortować

[]	← Pusta tablica
[20]	← Tablica z jednym elementem



Tablice puste i zawierające tylko jeden element będą naszym przypadkiem podstawowym. Można je zwracać w takim samym stanie, w jakim zostały przekazane do funkcji — nie ma w nich nic do sortowania.

```
def quicksort(array):
    if len(array) < 2:
        return array
```

Teraz przyjrzymy się większym tablicom. Tablica zawierająca dwa elementy też nie sprawi problemów przy sortowaniu.

1	7
---	---

Sprawdź, czy pierwszy element
jest mniejszy od drugiego. Jeśli
nie, zamień elementy miejscami

A co z tablicą zawierającą trzy elementy?

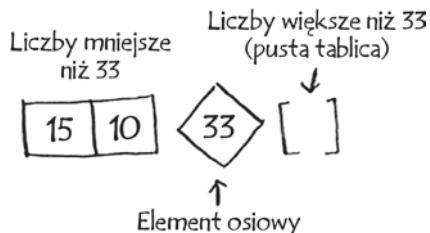
33	15	10
----	----	----

Przypomnę, że zamierzamy korzystać ze strategii „dziel i rządź”. Powinniśmy więc dzielić tę tablicę aż do osiągnięcia przypadku podstawowego. Oto, jak działa algorytm sortowania szybkiego. Najpierw wybieramy element z tablicy, tzw. **element osiowy** (ang. *pivot*).



Metody wybierania dobrego elementu osiowego opisuję później.
Na razie powiedzmy, że jest nim pierwszy element tablicy.

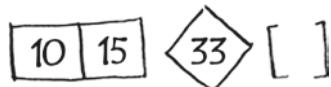
Teraz szukamy elementów, które są mniejsze i większe od osiowego.



Czynności przedstawione na tym rysunku nazywamy **partycjonowaniem** (ang. *partitioning*). Teraz mamy:

- podtablicę wszystkich liczb mniejszych od elementu osiowego,
- element osiowy,
- podtablicę wszystkich liczb większych od elementu osiowego.

Podtablice nie są posortowane, a jedynie zawierają część elementów tablicy. Gdyby jednak *były posortowane*, wtedy posortowanie całej tablicy nie sprawiłoby żadnej trudności.



Gdy podtablice są posortowane, wszystkie składniki można połączyć w jedną całość w następujący sposób: lewa tablica + element osiowy + prawa tablica. W ten sposób otrzymujemy posortowaną tablicę. W tym przypadku należałoby wykonać następującą operację: $[10, 15] + 33 + [] = [10, 15, 33]$, wynikiem której jest posortowana tablica.

A jak posortować podtablice? Przypadek podstawowy algorytmu sortowania szybkiego określa już, jak posortować tablice zawierające dwa elementy (lewa podtablica) i puste (prawa podtablica). Jeśli więc wywołamy funkcję `quicksort` na obu podtablicach i połączymy wyniki, otrzymamy tablicę posortowaną!

```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33] <----- Tablica posortowana.
```

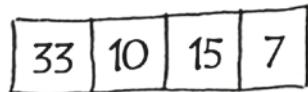
Technika ta zadziała z każdym elementem osiowym. Powiedzmy, że jest nim liczba 15, a nie 33.



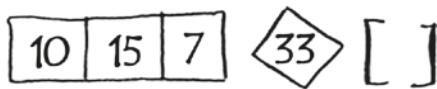
Obie podtablice mają po jednym elemencie, zatem wiadomo, jak je posortować. Wiemy więc już, jak posortować tablicę trzech elementów. Oto lista czynności, jakie należy wykonać.

1. Wybierz element osiowy.
2. Podziel tablicę na dwie podtablice — z elementami mniejszymi i większymi od osiowego.
3. Rekurencyjnie wywołuj funkcję sortującą na obu podtablicach.

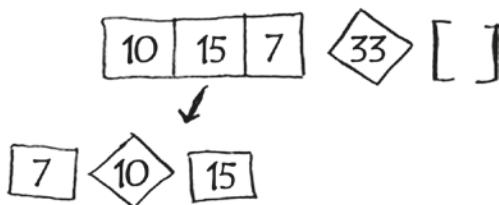
A gdyby tablica zawierała cztery elementy?



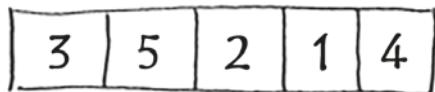
Powiedzmy, że na element osiowy ponownie wybieramy 33.



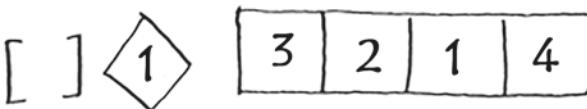
Tablica po lewej stronie zawiera trzy elementy. Już wiesz, jak posortować tablicę trzech elementów — należy na niej rekurencyjnie wywołać funkcję sortowania szybkiego.



Umiemy więc też posortować tablicę czterech elementów. A skoro tak, to po-
radzimy sobie też ze strukturą pięcioelementową. W jaki sposób? Powiedzmy,
że mamy do posortowania poniższą tablicę pięciu elementów.

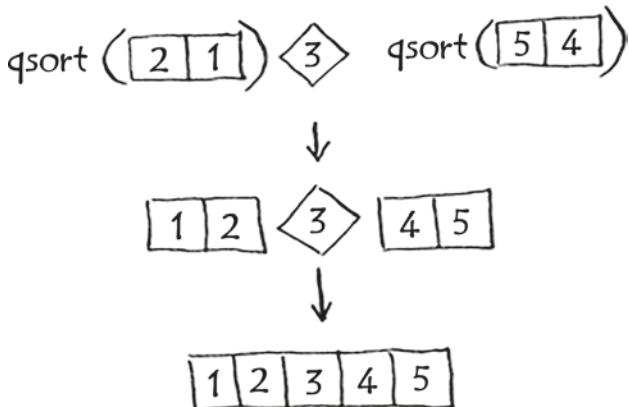


Oto wszystkie możliwe sposoby jej partycjonowania w zależności od wyboru
elementu osiowego.

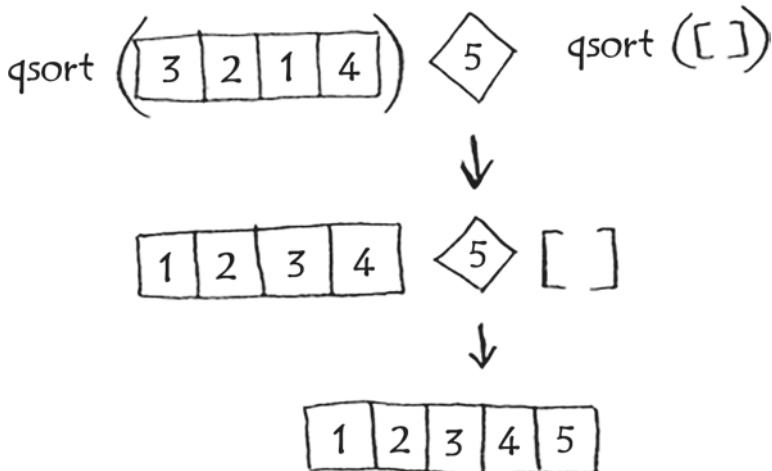


Każda z podtablic zawiera od zera do czterech elementów. A my już umiemy
sortować tablice tych rozmiarów przy użyciu algorytmu sortowania szybkie-
go! Zatem nieważne, jaki element osiowy wybierzemy, na powstałych pod-
tablicach możemy rekurencyjnie wywołać funkcję sortowania.

Powiedzmy np., że na element osiowy wybieramy trójkę. Wówczas wywołujemy funkcję sortowania szybkiego na następujących podtablicach.



Podtablice zostają posortowane, po czym składamy wyniki w jedną posortowaną tablicę. Operacja uda się nawet wtedy, gdy na element osiowy wybierzemy piątkę.



Metoda ta zadziała zawsze, bez względu na to, która liczba zostanie wybrana na element osiowy. Umiemy więc posortować tablicę pięciu elementów. Analogicznie możemy posortować tablicę sześciu elementów itd.

Dowody indukcyjne

Właśnie niepostrzeżenie przeczytałeś wprowadzenie do **dowodów indukcyjnych**! Jest to jedna z metod udowadniania, że dany algorytm działa. Każdy dowód indukcyjny składa się z dwóch części: przypadku podstawowego i przypadku indukcyjnego. Brzmi znajomo? Wyobraź sobie np., że chcę udowodnić, iż umiem wejść na ostatni szczebel drabiny. W przypadku indukcyjnym, jeśli moje nogi znajdują się na jakimś szczeblu, to mogę je postawić na następnym szczeblu. Jeżeli więc stoję na szczeblu drugim, mogę wejść na trzeci. To jest przypadek indukcyjny. A jeśli chodzi o przypadek podstawowy, mogę powiedzieć, że moje nogi są na szczeblu pierwszym. W związku z tym mogę wejść na samą góre, wspinając się po jednym szczeblu na raz.

Podobne rozumowanie stosuje się w sortowaniu szybkim. Najpierw wykałem, że algorytm działa w przypadku podstawowym, czyli dla tablic o długości 0 i 1. W przypadku indukcyjnym pokazałem, że jeśli sortowanie szybkie działa dla tablicy o rozmiarze 1, to będzie działać też dla tablicy o rozmiarze 2. A skoro działa dla tablicy o rozmiarze 2, to zadziała i dla tablicy o rozmiarze 3 itd. Potem mogę stwierdzić, że sortowanie szybkie zadziała dla tablic każdego rozmiaru. Nie będę dalej wchodził w dowodzenie indukcyjne, ale podkreśli, że jest bardzo zabawne i idzie w parze ze strategią „dziel i rządź”.

Oto przykładowa implementacja algorytmu sortowania szybkiego.

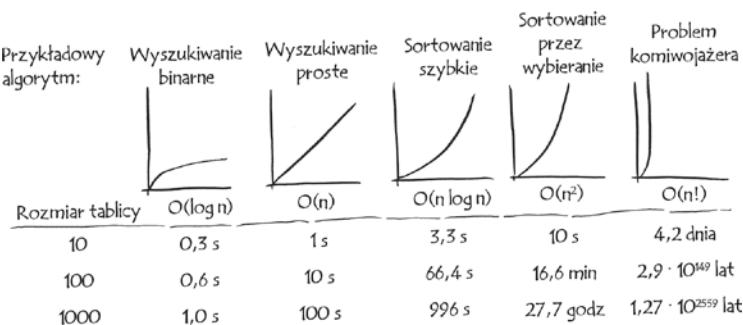
```
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
print(quicksort([10, 5, 2, 3]))
```



Przypadek podstawowy: tablice puste i jednoelementowe są góry „posortowane”. Podtablica zawierająca wszystkie elementy mniejsze od elementu osiowego. Podtablica zawierająca wszystkie elementy większe od elementu osiowego.

Jeszcze raz o notacji dużego O

Sortowanie szybkie to niezwykły algorytm, ponieważ jego szybkość zależy od wyboru elementu osiowego. Zanim omówię ten algorytm, jeszcze raz spojrzymy na typowe czasy działania różnych algorytmów.



Szacunkowe obliczenia wykonane dla wolnego komputera wykonującego 10 operacji na sekundę.

Przedstawione wartości obliczono przy założeniu, że komputer jest w stanie wykonywać 10 operacji na sekundę. Wykresy nie są dokładne — przedstawiłem je tylko po to, aby pokazać, jak bardzo algorytmy różnią się między sobą pod względem wydajności. Prawdziwe komputery wykonują znacznie więcej niż 10 operacji na sekundę.

Przy każdym czasie wykonywania podany został też przykładowy algorytm. Spójrz np. na wybieranie przez sortowanie, które opisałem w rozdziale 2. Jego czas wykonywania wynosi $O(n^2)$, co oznacza, że jest to bardzo powolny algorytm.

Istnieje jeszcze inny algorytm sortowania, o nazwie **sortowanie przez scalanie** (ang. *merge sort*), który charakteryzuje się wydajnością $O(n \log n)$. To znacznie lepszy wynik! Sortowanie szybkie jest natomiast trudnym przypadkiem. W najgorszym razie jego czas wykonywania wynosi $O(n^2)$.

Znaczy to, że sortowanie szybkie jest tak samo wolne jak sortowanie przez wybieranie. Jednak tylko w najgorszym przypadku. W średnim przypadku wydajność wynosi $O(n \log n)$. Oto pytania, nad którymi pewnie się zastanawiasz.

- Co znaczą **najgorszy przypadek i średni przypadek**?
- Skoro średnia wydajność sortowania szybkiego wynosi $O(n \log n)$, a sortowanie przez scalanie ma taką wydajność zawsze, dlaczego nie używa się zawsze tego drugiego algorytmu? Czy nie jest szybszy?

Sortowanie przez scalanie a sortowanie szybkie

Spójrz na poniższą prostą funkcję drukującą wszystkie elementy listy.

```
def print_items(list):
    for item in list:
        print item
```

Funkcja ta przegląda wszystkie elementy listy po kolej i je drukuje. Ponieważ pętla przechodzi przez całą listę raz, złożoność obliczeniowa tej funkcji wynosi $O(n)$. A teraz pomyśl sobie, że dodajesz do tej funkcji instrukcję usypiającą ją na sekundę przed wydrukowaniem każdego elementu.

```
from time import sleep
def print_items2(list):
    for item in list:
        sleep(1)
        print item
```

Przed wydrukowaniem wartości funkcja wstrzymuje działanie na sekundę. Powiedzmy, że drukujemy listę pięciu elementów przy użyciu każdej z tych funkcji.

2	4	6	8	10
---	---	---	---	----

↓

print_items: 2 4 6 8 10
print_items2: <sleep> 2 <sleep> 4 <sleep> 6 <sleep> 8 <sleep> 10

Obie funkcje przeglądają listę raz, więc obie mają złożoność obliczeniową $O(n)$. Która z nich, Twoim zdaniem, będzie szybsza w praktyce? Uważam, że znacznie szybsza będzie funkcja `print_items`, ponieważ nie wstrzymuje działania na sekundę przed wydrukowaniem każdego elementu. W związku z tym, mimo że obie funkcje są tak samo szybkie w notacji dużego O, w rzeczywistości funkcja `print_items` jest szybsza. Dlatego zapis $O(n)$ w notacji dużego O naprawdę oznacza to:

$$c^* n$$

Jakaś stała \nearrow
 ilość czasu

Parametr c oznacza stałą ilość czasu wykorzystywaną przez algorytm, zwaną w skrócie **stałą**. Dla funkcji `print_items` może to być np. *10 milisekund * n*, a dla funkcji `print_items2` — *1 sekunda * n*.

Stałą zazwyczaj się pomija, ponieważ nie ma ona znaczenia, jeśli dwa algorytmy mają różne czasy wykonywania w notacji dużego O. Weźmy np. wyszukiwanie binarne i proste. Założymy, że algorytmy te mają następujące stałe.

$$\frac{10 \text{ ms} * n}{\text{Wyszukiwanie proste}} \quad \frac{1 \text{ s} * \log n}{\text{Wyszukiwanie binarne}}$$

Można powiedzieć: „Wow! Wyszukiwanie proste ma stałą tylko 10 milisekund, a binarne aż sekundę. Wyszukiwanie proste jest bez porównania szybsze!”. Teraz wyobraź sobie, że przeszukujesz listę 4 miliardów elementów. Oto czas potrzebny na wykonanie tych operacji.

Wyszukiwanie proste	$10 \text{ ms} * 4 \text{ miliardy} = 436 \text{ dni}$
Wyszukiwanie binarne	$1 \text{ s} * 32 = 32 \text{ sekundy}$

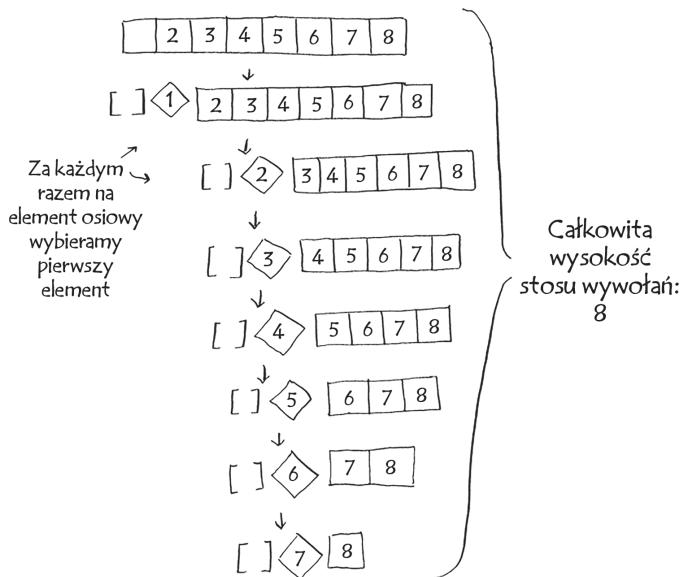
Jak widać, wyszukiwanie binarne jest znacznie szybsze. Wielkość stałej nie odegrała żadnej roli.

Czasami jednak stała *może robić różnicę*. Przykładem takiej sytuacji jest porównanie wydajności algorytmów sortowania przez scalanie i sortowania szybkiego. Ten drugi ma stałą o niższej wartości. Jeśli więc oba algorytmy mają złożoność obliczeniową na poziomie $O(n \log n)$, to sortowanie szybkie jest wydajniejsze. I rzeczywiście jest tak też w praktyce, ponieważ średni przypadek zdarza się o wiele częściej niż najgorszy.

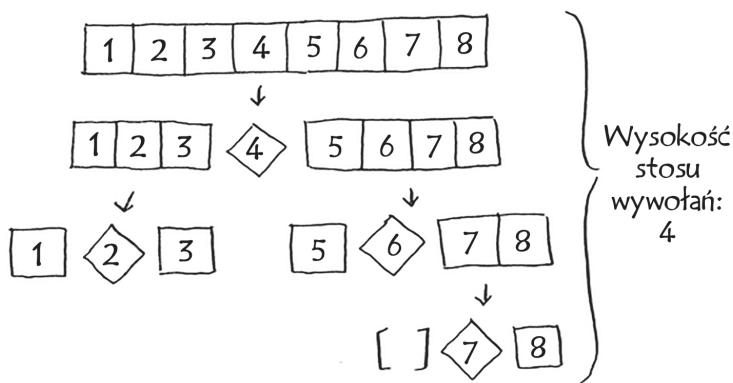
Teraz pewnie się zastanawiasz, jaka jest różnica między przypadkami średnim i najgorszym?

Przypadki średni i najgorszy

Wydajność algorytmu sortowania szybkiego w znacznym stopniu zależy od wyboru elementu osiowego. Powiedzmy, że na element osiowy zawsze wybierasz pierwszą wartość i wywołujesz sortowanie szybkie z tablicą, która jest *już posortowana*. Algorytm nie sprawdza, czy tablica jest posortowana, więc będzie próbował ją posortować.



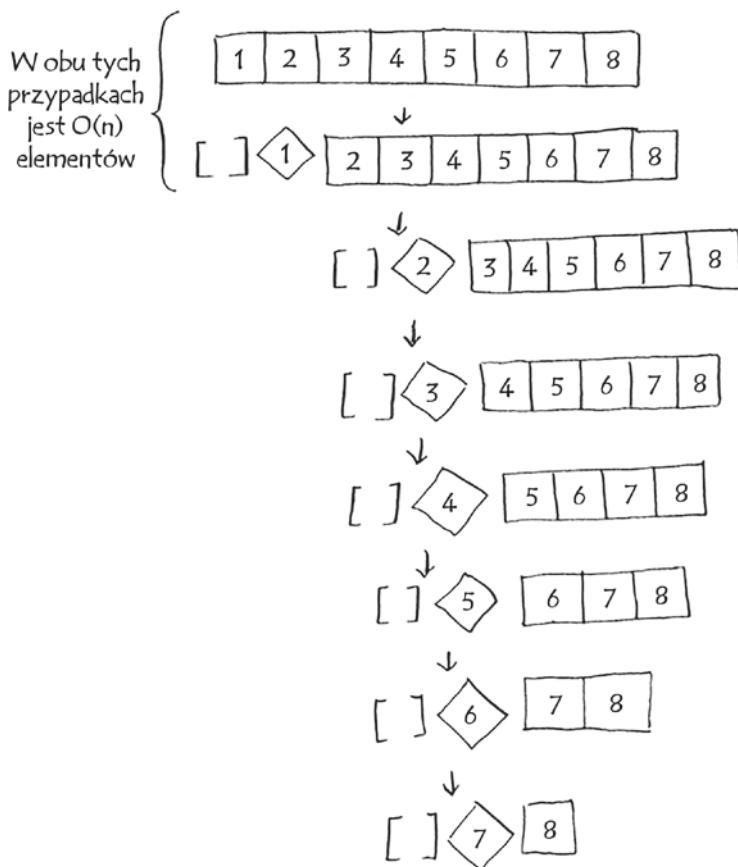
Zauważ, że tablica ani razu nie jest dzielona na pół. Zamiast tego jedna z podtablic zawsze jest pusta, przez co stos wywołań jest bardzo wysoki. A teraz wyobraź sobie, że zawsze wybierasz środkowy element na element osiowy. Spójrz teraz na wysokość stosu wywołań.



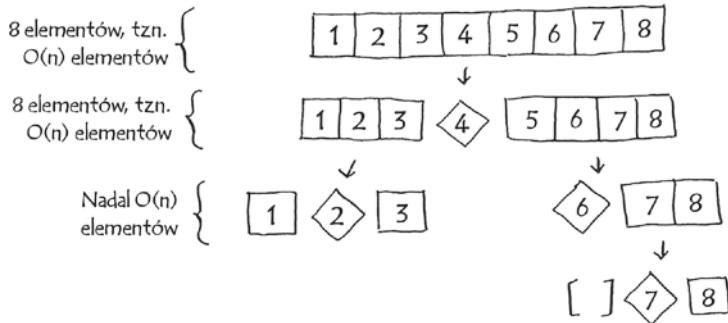
Stos jest bardzo niski! Ponieważ za każdym razem tablicę dzielimy na pół, nie musimy wykonywać tak wielu wywołań rekurencyjnych. Szybciej docieramy do przypadku podstawowego i dzięki temu nasz stos wywołań znacznie się zmniejsza.

Pierwszy z przedstawionych przykładów to właśnie najgorszy przypadek, a drugi — najlepszy. W najgorszym przypadku rozmiar stosu wynosi $O(n)$. W najlepszym przypadku rozmiar ten wynosi $O(\log n)$.

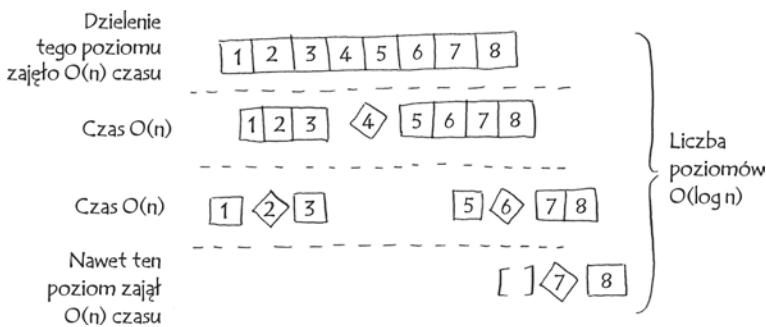
A teraz spojr자 na pierwszy poziom w stosie. Wybierasz jeden element na element osiowy, a pozostałe elementy dzielisz na dwie podtablice. Dotykamy wszystkich ośmiu elementów w tablicy, więc pierwsza operacja zajmuje $O(n)$ czasu. Dotknęliśmy wszystkich ośmiu elementów na tym poziomie stosu. Jednak w rzeczywistości dotykamy $O(n)$ elementów na każdym jego poziomie.



Gdybyśmy nawet inaczej podzieliли tablicę, nadal za każdym razem dotyczyćlibyśmy $O(n)$ elementów.



Zatem wykonanie każdego poziomu zajmuje $O(n)$ czasu.



W tym przykładzie jest $O(\log n)$ poziomów (choć fachowo powinno się powiedzieć: „Wysokość stosu wywołań wynosi $O(\log n)$ ”). Każdy poziom zajmuje $O(n)$ czasu. Zatem złożoność obliczeniowa całego algorytmu to $O(n) \cdot O(\log n) = O(n \log n)$. Tak jest w najlepszym przypadku.

W najgorszym przypadku jest $O(n)$ poziomów, a więc wykonywanie algorytmu zajmie $O(n) \cdot O(n) = O(n^2)$ czasu.

Wiesz co? Chciałbym poinformować, że tutaj najlepszy przypadek jest jednocześnie średnim przypadkiem. Jeśli na element osiowy zawsze będziesz wybierać losowy element z tablicy, to średni czas sortowania szybkiego będzie wynosił $O(n \log n)$. Sortowanie szybkie to jeden z najszybszych algorytmów sortowania, który dodatkowo jest doskonałym przykładem zastosowania strategii „dziel i rządź”.

ĆWICZENIA

Przedstaw czas wykonywania wszystkich poniższych operacji w notacji dużego O.

- 4.5. Drukowanie wartości wszystkich elementów w tablicy.
- 4.6. Podwojenie wartości każdego elementu w tablicy.
- 4.7. Podwojenie wartości tylko pierwszego elementu tablicy.
- 4.8. Utworzenie tabliczki mnożenia dla wszystkich elementów w tablicy. Jeśli np. dana jest tablica $[2, 3, 7, 8, 10]$, to najpierw mnożysz wszystkie elementy przez 2, potem przez 3, potem przez 7 itd.

Powtórzenie

- Strategia „dziel i rządź” polega na dzieleniu problemu na coraz mniejsze części. Jeśli chodzi o listy, to przypadkiem podstawowym najczęściej jest tablica pusta lub zawierająca jeden element.
- Implementując algorytm sortowania szybkiego, element osiowy należy wybierać losowo. Średni czas wykonywania tego algorytmu wynosi $O(n \log n)!$.
- Stała w notacji dużego O czasami ma znaczenie. Dlatego algorytm sortowania szybkiego jest szybszy od sortowania przez scalanie.
- Wielkość stałej prawie nigdy nie ma znaczenia przy porównywaniu wyszukiwania prostego z binarnym, ponieważ w przypadku długich list między $O(\log n)$ i $O(n)$ jest zbyt duża różnica.





W tym rozdziale:

- poznasz tablice skrótów, które są jedną z fundamentalnych struktur danych. Tablice te znajdują wiele zastosowań. W rozdziale przedstawiam najczęstsze z nich.
- poznasz wewnętrzne mechanizmy działania tablic skrótów. Dowiesz się, jak są zaimplementowane, na czym polegają kolizje oraz co to są funkcje obliczania skrótów. Korzystając z tej wiedzy, będziesz w stanie analizować wydajność tablic skrótów.

Wyobraź sobie, że jesteś sprzedawcą w zieleniaku. Kiedy klient robi zakupy, musisz sprawdzić ceny produktów w cenniku. Jeśli lista jest nieposortowana alfabetycznie, znalezienie każdej pozycji dotyczącej *jabłek* może zająć Ci dużo czasu. Musiałbyś przeprowadzić opisane w rozdziale 1. wyszukiwanie proste. Pamiętasz, ile trwają takie operacje? Czas wyszukiwania prostego wynosi $O(n)$. Gdyby cennik był posortowany alfabetycznie, cenę jabłek można by znaleźć za pomocą wyszukiwania binarnego, które zajmuje tylko $O(\log n)$ czasu.

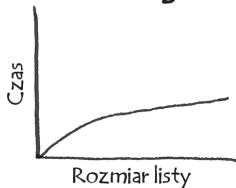
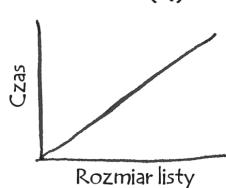


gruszka... 0,79 zł
jaja..... 2,49 zł
mleko..... 1,99 zł

Lista posortowana

jaja..... 2,49 zł
mleko..... 1,99 zł
gruszka... 0,79 zł

Lista nieposortowana

 $O(\log n)$  $O(n)$ 

Przypomnę, że między $O(n)$ i $O(\log n)$ jest duża różnica! Powiedzmy, że jesteś w stanie przejrzeć 10 pozycji cennika na sekundę. Poniżej znajduje się zestawienie czasów wyszukiwania prostego i binarnego przy tym założeniu.

Liczba pozycji w cenniku	$O(n)$	$O(\log n)$
100	10 s	1 s ← Trzeba sprawdzić $\log_2 100 = 7$ pozycji
1000	1,66 min	1 s ← Trzeba sprawdzić $\log_2 1000 = 10$ pozycji
10 000	16,6 min	2 s ↵ $\log_2 10\,000 = 14$ pozycji = 2 s

Wiesz już, że wyszukiwanie binarne jest niesamowicie szybkie. Jednak dla kasjera szukanie cen w cenniku jest kłopotliwe nawet wtedy, kiedy pozycje są posortowane alfabetycznie. Dosłownie czujesz, jak klient zaczyna się gotować, gdy Ty skrupulatnie przeglądasz listę. Przydałby się pomocnik znający na pamięć wszystkie nazwy i ceny. Wówczas nie musiałbyś wszystkiego szukać — wystarczy zapytać pomocnika, a ten od razu odpowie.



Twoja pomocnica Magda dostarcza ceny w czasie $O(1)$, niezależnie od wielkości cennika. Jest więc nawet szybsza niż wyszukiwanie binarne.

Liczba pozycji w cenniku	Wyszukiwanie proste	Wyszukiwanie binarne	Magda
	$O(n)$	$O(\log n)$	$O(i)$
100	10 s	1 s	Natychmiast
1000	1,66 min	1 s	Natychmiast
10 000	16,6 min	2 s	Natychmiast

Co za cudowna osoba! Skąd wziąć taką „Magdę”?

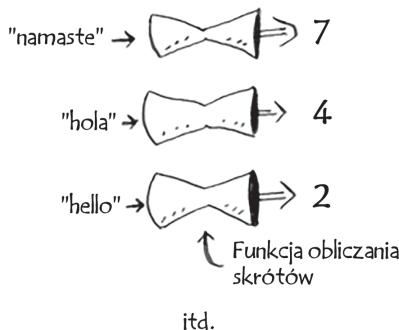
Przywdziejmy po raz kolejny szaty specjalistów od struktur danych. Znasz już dwie takie struktury, czyli tablice i listy (nie zaliczam do nich stosu, ponieważ nie można go „przeszukiwać”). Zatem omawiany cennik mógłbyś zaimplementować jako tablicę.

(jaja, 2.49)	(mleko, 1.49)	(gruszka, 0.79)
--------------	---------------	-----------------

Każdy element tablicy składa się z dwóch elementów — nazwy produktu i jego ceny. Jeśli posortujemy tę tablicę wg nazw, za pomocą wyszukiwania binarnego będziemy mogli w niej znajdować ceny produktów. Zatem możemy znajdować produkty w czasie $O(\log n)$. Jednak nam zależy na czasie wyszukiwania rzędu $O(1)$. Innymi słowy, chcemy mieć „Magdę”. Do tego potrzebne będą funkcje obliczania skrótów (ang. *hash function*).

Funkcje obliczania skrótów

Funkcja obliczania skrótów pobiera łańcuch¹ i zwraca liczbę.

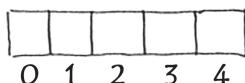


Mówiąc bardziej fachowo, funkcja obliczania skrótów „odwzorowuje łańcuchy jako liczby”. Możesz sobie pomyśleć, że liczby dla poszczególnych łańcuchów są zwracane bez żadnej zauważalnej logiki. A jednak funkcjom obliczania skrótów stawia się pewne wymagania.

- Funkcja musi być stabilna, tzn. jeśli raz przekaże się jej napis „jabłko” i zwróci „4”, to za każdym następnym razem powinna zwrócić tę samą wartość dla tego wyrazu. Bez spełnienia tego warunku nie ma co marzyć o tablicy skrótów.
- Dla różnych słów funkcja musi generować różne liczby. Do niczego nie przyda się np. funkcja, która zawsze zwraca „1”. Najlepsza jest taka funkcja, która dla każdego słowa zwraca inną wartość.

A zatem funkcja obliczania skrótów odwzorowuje łańcuchy jako liczby. Jaki z tego pożytek? Przykładowo taki, że dzięki temu możemy stworzyć własną „Magdę”!

Zaczniemy od pustej tablicy.

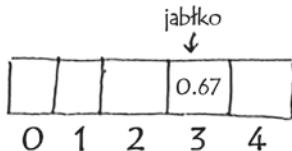


W strukturze tej zapiszemy wszystkie ceny naszych produktów. Zaczniemy od dodania ceny jabłek. W tym celu przekazujemy „jabłko” do funkcji obliczania skrótów.

1. W tym kontekście *łańcuch* to każdy rodzaj danych, a dokładniej po prostu sekwencja bajtów.

"jabłko" →  3

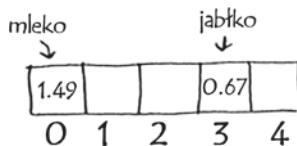
Funkcja zwraca liczbę „3”, więc zapisujemy cenę jabłka pod indeksem o takim numerze w tablicy.



Teraz dodamy mleko. W tym celu do funkcji przekazujemy napis „mleko”.

"mleko" →  0

Funkcja zwróciła „0”, więc zapisujemy cenę mleka pod indeksem 0.



Kontynuując pracę w ten sposób, wkrótce otrzymamy tablicę pełną cen.

1.49	0.79	2.49	0.67	1.49
------	------	------	------	------

Teraz pewnie sobie myślisz: „Ciekawe, ile kosztuje awokado”? Nie musisz szukać tego owocu w tablicy. Wystarczy, że przekażesz łańcuch „awokado” do funkcji obliczania skrótów.

"awokado" →  4

W ten sposób dowiesz się, że cena znajduje się pod indeksem 4. Gdy sprawdzisz, okaże się, że to prawda.

$$\text{awokado} = 1.49$$

1.49	0.79	2.49	0.67	1.49
/	/	/	/	\

Funkcja obliczania skrótów informuje, gdzie dokładnie znajduje się cena danego produktu, więc wcale nie trzeba jej szukać! Jest to możliwe, ponieważ funkcja obliczania skrótów zachowuje się wg pewnych zasad. Oto one.

- Funkcja obliczania skrótów konsekwentnie odwzorowuje tę samą nazwę zawsze jako ten sam indeks. Za każdym razem, gdy przekażemy jej napis „awokado”, zwrotnie otrzymamy tę samą liczbę. Zatem najpierw przy użyciu tej funkcji znajdziemy miejsce do zapisania ceny awokado, a następnie wykorzystujemy ją do znalezienia miejsca przechowywania tej informacji.
- Funkcja obliczania skrótów odwzorowuje różnełańcuchy jako różne indeksy. Słowo „awokado” zostaje przypisane do indeksu 4. Słowo „mleko” otrzymuje indeks 0. Każda kolejna pozycja trafia do innej komórki tablicy, w której zostaje zapisana cena.
- Funkcja obliczania skrótów „zna” rozmiar tablicy i zwraca tylko prawidłowe indeksy. Jeśli więc tablica zawiera pięć elementów, funkcja nie zwróci indeksu 100, ponieważ w tej strukturze byłby on nieprawidłowy.

Właśnie stworzyliśmy „Magdę”! Wystarczy połączyć funkcję obliczania skrótów i tablicę, aby otrzymać strukturę danych zwaną **tablicą skrótów** (ang. *hash table*) — zapewne częściej spotkasz się z określeniem „tablica mieszająca”, ale nie oddaje ono w pełni charakteru tej struktury. Jest to pierwsza struktura danych, która ma wbudowaną dodatkową logikę. Tablice i listy odnoszą się wprost do pamięci. Natomiast tablice skrótów są inteligentniejsze, ponieważ obliczają miejsce zapisu danych za pomocą specjalnej funkcji.

Niewykluczone, że tablice skrótów będą najbardziej przydatną złożoną strukturą danych, jaką poznasz. Jednak przede wszystkim tablice skrótów są niesamowicie szybkie! Pamiętasz opis tablic i list powiązanych z rozdziałem 2.? Z tablicy element można pobrać błyskawicznie. A tablice skrótów przechowują dane właśnie w tablicach, więc są również szybkie.

Prawdopodobnie nigdy nie będziesz musiał samodzielnie implementować tablice skrótów. Każdy dobry język programowania ma już gotową implementację tej struktury danych. W Pythonie występuje ona pod nazwą **słownik** (ang. *dictionary*). Nową tablicę skrótów można utworzyć przy użyciu funkcji `dict`.

```
>>> book = dict()
```



Pusta tablica skrótów

Za pomocą powyższej instrukcji utworzyliśmy tablicę skrótów o nazwie book. Teraz dodamy do niej ceny kilku produktów.

```
>>> book["jabłko"] = 0.67 ←..... Jabłko kosztuje 67 groszy.  
>>> book["mleko"] = 1.49 ←..... Mleko kosztuje 1,49 zł.  
>>> book["awokado"] = 1.49  
>>> print book  
{'awokado': 1.49, 'jabłko': 0.67, 'mleko': 1.49}
```

jabłko	0.67
mleko	1.49
awokado	1.49

Tаблица скротов
з ценами продуктow

Łatwizna! Teraz poprośmy o cenę awokado.

```
>>> print book["awokado"]  
1.49 ←..... Cena awokado.
```

Tablica skrótów zawiera klucze i wartości. W tablicy book kluczami są nazwy produktów, a wartościami — ich ceny. Tablica skrótów przypisuje kluczom wartości.

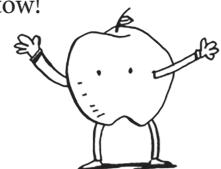
W kolejnym podrozdziale opisuję kilka przykładów praktycznego wykorzystania tych struktur danych.

ĆWICZENIA

Bardzo ważną cechą funkcji obliczania skrótów jest zwracanie zawsze takiego samego wyniku dla tych samych danych wejściowych. Gdyby funkcja nie spełniała tego warunku, nie dałoby się znaleźć wstawionych do tablicy elementów!

KTÓRE Z PONIŻSZYCH FUNKCJI SĄ KONSEKVENTNE W TEJ KWESTII?

- 5.1.** $f(x) = 1$ ←..... Zwraca „1” dla wszystkich danych wejściowych.
- 5.2.** $f(x) = \text{rand}()$ ←..... Za każdym razem zwraca losową liczbę.
- 5.3.** $f(x) = \text{next_empty_slot}()$ ←..... Zwraca indeks następnej pustej komórki w tablicy skrótów.
- 5.4.** $f(x) = \text{len}(x)$ ←..... Jako indeksu używa długości łańcucha.



Zastosowania tablic skrótów

Tablice skrótów mają bardzo szerokie spektrum zastosowań. W tym podroziale przedstawiam tylko niektóre z nich.

Przeszukiwanie tablic skrótów

Twój telefon zawiera bardzo wygodną książkę telefoniczną, w której każdemu nazwisku przypisany jest numer telefoniczny.

Bade Mama → 581 660 9820

Alex Manning → 484 234 4680

Jane Marin → 415 567 3579



Wyobraź sobie, że chcesz utworzyć taką książkę. Przypisujesz nazwiskom numery telefonu, więc Twój program musi mieć następujące funkcje.

- Dodawanie nazwisk osób i wiązanie z nimi numerów telefonicznych.
- Wyszukiwanie numeru telefonicznego skojarzonego z wpisanym nazwiskiem.

To idealny przypadek do wykorzystania tablicy skrótów. Struktury te są doskonale do:

- przypisywania jednej porcji informacji do innej,
- wyszukiwania.

Książka telefoniczna ma bardzo prostą budowę. Zacząć należy od utworzenia nowej tablicy skrótów.

```
>>> phone_book = dict()
```

Przy okazji, w Pythonie istnieje skrót do tworzenia tablic skrótów w postaci klamry.

```
>>> phone_book = {} <----- To samo, co phone_book = dict().
```

Teraz możemy dodać do naszej książki numery kilku osób.

```
>>> phone_book["jasia"] = 8675309
>>> phone_book["pogotowie"] = 112
```

Jasia	8675309
Pogotowie	112

Tablica skrótów jako książka telefoniczna

```
>>> print phone_book["jasia"]
8675309 <----- Numer telefonu Jasi.
```

To wszystko! Wyobraź sobie, że chcesz sprawdzić numer Jasi. W tym celu tylko przekazujesz klucz do tablicy.

Wyobraź sobie, że jednak musisz skorzystać z tablicy. Co wtedy zrobisz? Tablice skrótów bardzo ułatwiają modelowanie relacji między elementami.

Tablicy skrótów najczęściej używa się w znacznie większej skali. Powiedzmy np., że wchodzimy na stronę <http://adit.io>. Nasz komputer musi przetłumaczyć domenę *adit.io* na adres IP.

adit.io → 173.255.245.55

Przy wizycie na każdej stronie internetowej konieczne jest przetłumaczenie adresu internetowego na adres IP.

google.com → 74.125.239.133

facebook.com → 173.252.120.6

scribd.com → 23.235.47.175

Wow, oto odwzorowanie nazwy domeny jako adresu IP! Wygląda na idealne zastosowanie dla tablic skrótów! Ten proces translacji nazywa się **rozpoznaniem nazw DNS**, a tablice skrótów są jednym z filarów implementacji tego mechanizmu.

Zapobieganie powstawaniu duplikatów elementów

Wyobraź sobie, że nadzorujesz kabinę do głosowania. Oczywiście każdy może oddać tylko jeden głos. Jak sprawdzić, czy dana osoba już wcześniej nie głosowała? Kiedy ktoś przychodzi, aby zagłosować, prosisz o podanie imienia i nazwiska. Następnie sprawdzasz, czy nie ma ich na liście tych, którzy oddali głos.



Jeśliczyjeś nazwisko znajduje się na liście, wiadomo, że dana osoba już głosowała i należy ją pogonić! W przeciwnym razie dodajemy nowe nazwisko do listy i umożliwiamy oddanie głosu. A teraz wyobraź sobie, że do głosowania zgłosiło się bardzo dużo osób i lista osób, które już zagłosowały, jest bardzo długa.



Za każdym razem, gdy ktoś przyjdzie zagłosować, musisz przejrzeć gigantyczną listę nazwisk osób, które już głosowały. A przecież jest lepsze rozwiązanie — wystarczy użyć tablicy skrótów!

Najpierw utworzymy tablicę skrótów do przechowywania danych osób, które już oddały głos.

```
>>> voted = {}
```

Gdy zgłosi się nowa osoba, sprawdzamy, czy jest już zarejestrowana w tablicy.

```
>>> value = voted.get("tomasz")
```

Jeśli w tablicy skrótów znajduje się pozycja o nazwie "tomasz", funkcja get zwraca wartość. Jeśli jej nie ma, zwraca None. Można to wykorzystać do sprawdzenia, czy wybrana osoba już zagłosowała!

Oto odpowiedni kod źródłowy.

```
voted = {}
def check_voter(name):
    if voted.get(name):
        print "Pogonić go!"
    else:
        voted[name] = True
        print "Niech zagłosuje!"
```



Sprawdźmy, czy to działa.

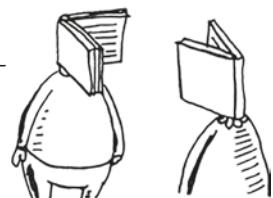
```
>>> check_voter("tomasz")
Niech zagłosuje!
>>> check_voter("michał")
Niech zagłosuje!
>>> check_voter("michał")
Pogonić go!
```

Za pierwszym razem, gdy Tomasz zgłosi się do głosowania, program wydrukuje napis: Niech zagłosuje!. Następnie przychodzi Michał, któremu za pierwszym razem również pozwalamy zagłosować. Gdy jednak Michał spróbuje oddać głos po raz drugi, program wyświetli napis: Pogonić go!.

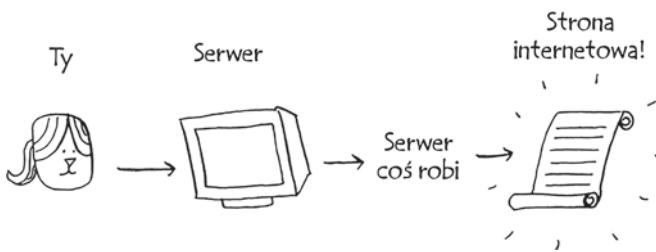
Gdybyśmy nazwiska osób, które już oddały głos, zapisywali w zwykłej liście, z czasem nasza funkcja stałaby się bardzo powolna, ponieważ musiałaby przeszukiwać za pomocą algorytmu wyszukiwania prostego całą listę. Jeżeli jednak będziemy zapisywać nazwiska w tablicy skrótów, ta błyskawicznie poinformuje nas, czy dane nazwisko jest już na liście, czy nie. W tablicach skrótów duplikaty wyszukuje się naprawdę szybko.

Tablice skrótów jako pamięć podręczna

Oto jeszcze jedno zastosowanie tablic skrótów — implementacja pamięci podręcznej. Każdy, kto tworzy strony internetowe, wie, że warto zapisywać dane w pamięci podręcznej. Już wyjaśniam, na czym to polega. Powiedzmy, że wchodzisz na stronę facebook.com.



1. Wysydasz żądanie do serwera Facebooka.
2. Serwer zastanawia się chwilę i przygotowuje stronę do wysłania do Twojej przeglądarki.
3. Otrzymujesz stronę internetową.

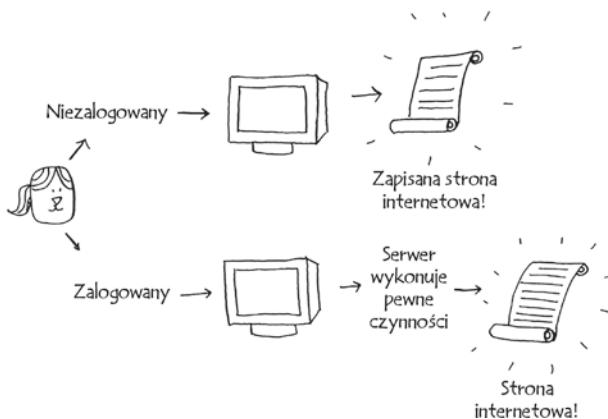


Serwer Facebooka może np. zapisywać wszystkie czynności Twojego znajomego, aby Ci je pokazać. Zbiera je przez kilka sekund, a następnie wyświetla informacje na stronie. Użytkownikowi te parę sekund może wydawać się wiecznością. W końcu zaczniesz się zastanawiać: „Dlaczego ten Facebook jest taki powolny?”. A przecież serwery tego portalu obsługują miliony użytkowników i tych kilka sekund wynika właśnie z dużej liczby danych do przejrzenia. Serwery mają bardzo dużo pracy przy obsłudze wszystkich żądań. Czy da się jakoś przyspieszyć Facebook i choć trochę odciążyć jego serwery?

Powiedzmy, że masz bratanicę, która zamęcza Cię pytaniami o planety. „Ile jest kilometrów od Ziemi do Marsa?”, „Jak daleko jest do Księżyca?”, „Ile jest kilometrów do Jowisza?”. Za każdym razem, gdy usłyszysz jedno z tych pytań, musisz poszukać informacji w internecie. To zajmuje kilka minut. A teraz

wyobraź sobie, że Twoja siostrzenica cały czas pyta tylko: „Ile jest kilometrów do Księżyca?”. Bardzo szybko zapamiętasz, że ta odległość wynosi 384 400 kilometrów i nie będziesz musiał ciągle tego sprawdzać. Innymi słowy, odpowiedzi będziesz podawać z pamięci. Tak też działa pamięć podręczna — na witrynach internetowych dane są zapamiętywane, aby nie trzeba było ciągle obliczać ich od nowa.

Jeśli ktoś zaloguje się na Facebook, portal prezentuje treść przygotowaną specjalnie dla niego. Za każdym razem, gdy wchodzisz na stronę facebook.com, serwery „zastanawiają się”, jaka treść będzie dla Ciebie najbardziej interesująca. A jeśli na stronę tę wejdzie użytkownik niezalogowany, ujrzy stronę logowania. Każdemu pokazywana jest taka sama strona. Facebook ciągle jest proszony o to samo: „Pokaż mi stronę główną dla niezalogowanego użytkownika”. Dlatego zamiast za każdym razem od nowa generować wygląd tej strony, portal zapamiętuje ją i błyskawicznie wysyła każdemu zainteresowanemu.



Nazywa się to **zapisywaniem w pamięci podręcznej** (ang. *caching*). Technika ta ma dwie zalety.

- Strony internetowe są ładowane znacznie szybciej z pomocą tego samego mechanizmu, który pozwala Ci szybciej podawać zapamiętaną odległość Ziemi od Księżyca. Gdy znów bratanica spyta o to, nie będziesz już musiał sprawdzać w Google, tylko podasz odpowiedź z pamięci.
- Serwery Facebooka mają mniej pracy.

Pamięć podręczną często wykorzystuje się w celu przyspieszenia działania różnych programów. Stosują ją wszystkie większe portale internetowe. A co najważniejsze, dane są zapisywane właśnie w tablicy skrótów!

Facebook buforuje nie tylko stronę główną, ale także strony *O Facebooku*, *Kontakt*, *Regulamin* i wiele innych. W związku z tym Facebook potrzebuje przypisania adresów URL do danych stron.

facebook.com/about → Dane strony „O Facebooku”

facebook.com → Dane strony głównej

Gdy ktoś otwiera stronę na Facebooku, serwis najpierw sprawdza, czy strona ta znajduje się w pamięci podręcznej.



Oto odpowiedni kod źródłowy.

```
cache = {}

def get_page(url):
    if cache.get(url):
        return cache[url] # Zwraca dane z pamięci podręcznej.
    else:
        data = get_data_from_server(url)
        cache[url] = data # Najpierw zapisuje dane w pamięci podręcznej.
    return data
```

Serwer musi pracować tylko wtedy, kiedy danego adresu URL nie ma w pamięci podręcznej. Zanim jednak dane zostaną zwrócone, są zapisywane w pamięci, aby można je było zwrócić następnym razem, gdy użytkownik wyśle do serwera takie same żądanie. W ten sposób przy następnej okazji serwer już nie będzie musiał wykonywać tych samych czynności.

Powtórzenie wiadomości

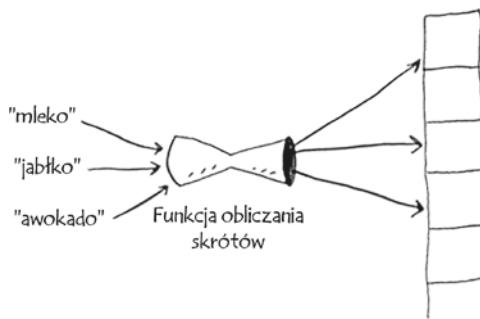
Oto podsumowanie. Tablice skrótów są dobre do:

- modelowania relacji między elementami,
- odfiltrowywania duplikatów,
- buforowania lub zapamiętywania danych, aby odciążyć serwery.

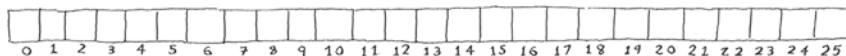
Kolizje

Jak napisałem wcześniej, większość języków programowania zawiera implementację tablic skrótów. Nie trzeba więc pisać jej samodzielnie. Dlatego też nie będę zbytnio rozwodził się nad wewnętrznymi mechanizmami działania tych struktur. Interesuje nas jednak wydajność, a żeby zrozumieć tę cechę tablic skrótów, należy wiedzieć, czym w ich kontekście są kolizje. Dlatego w dwóch następnych podrozdziałach opisuję kolizje i kwestię wydajności.

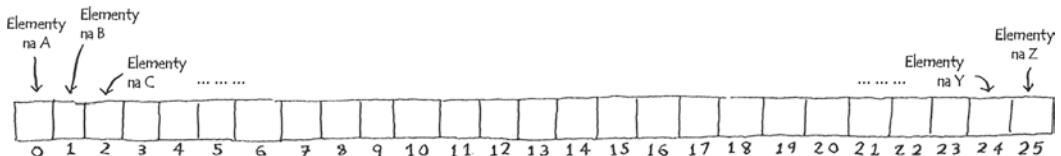
Najpierw przyznam się do drobnego kłamstwa. Napisalem, że funkcja obliczania skrótów zawsze przyporządkowuje różne klucze do różnych komórek w tablicy.

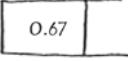


W rzeczywistości napisanie takiej funkcji jest prawie niemożliwe. Pomyśl np., że tablica zawiera 26 miejsc do przechowywania danych.

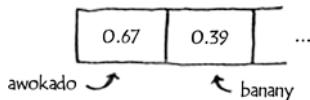


A nasza funkcja obliczania skrótów jest bardzo prosta, bo przypisuje elementom miejsca w tablicy wg alfabetu.

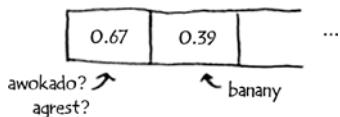


 ... Pewnie już się domyślasz, w czym rzecz. Gdy zechcesz zapisać cenę awokado w tej tablicy, otrzymasz do dyspozycji pierwszą komórkę.
awokado ↗

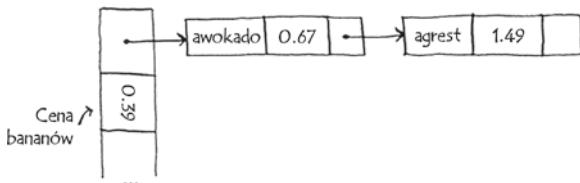
Następnie wstawiasz do tablicy cenę bananów, więc otrzymujesz drugą komórkę.



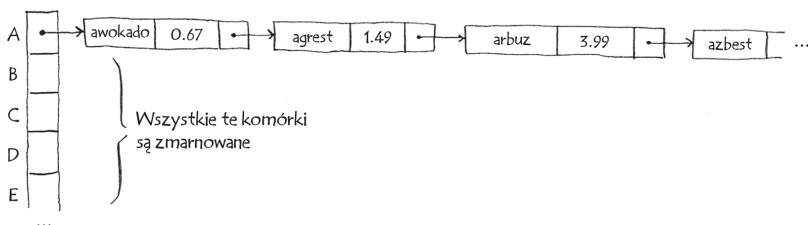
Wszystko idzie doskonale! A teraz chcesz dodać do tablicy cenę agrestu, więc znowu otrzymujesz pierwszą komórkę.



O nie! To miejsce jest już zajęte przez awokado! Co robić? Taka sytuacja nazywa się **kolizją**, gdyż dwa klucze zostały przypisane do jednego miejsca w tablicy. To jest problem. Jeśli zapiszemy tam cenę agrestu, nadpiszemy cenę awokado i gdy ktoś później poprosi o cenę awokado, to w rzeczywistości otrzyma cenę agrestu! Kolizje są złe i trzeba je jakoś wyeliminować. Jest wiele sposobów pozbycia się kolizji. Najprostszy polega na utworzeniu listy powiązanej w komórkach, do których przyporządkowano więcej niż jeden element.



W tym przykładzie awokado i agrest przynależą do tej samej komórki, więc tworzymy w niej listę powiązaną. Sprawdzanie ceny bananów nadal będzie bardzo szybką operacją, natomiast sprawdzenie ceny awokado zajmie już nieco więcej czasu, ponieważ będzie wymagało przeszukania listy powiązanej. Jeśli lista jest niedługa, nie ma problemu — przejrzenie trzech czy czterech elementów to żaden wysiłek. Jednak wyobraź sobie, że sklep, w którym pracujesz, sprzedaje tylko produkty na A.



Chwileczkę! Cała tablica skrótów jest pusta z wyjątkiem jednej komórki, w której znajduje się gigantyczna lista powiązana! Wszystkie elementy tej tablicy znajdują się w tej jednej liście. To tak samo słabe rozwiązańe, jakbyśmy od razu utworzyli zwykłą listę powiązaną. Wyszukiwanie w tej strukturze na pewno nie będzie tak szybkie, jakbyśmy tego sobie życzyli.

Nasuwały się dwa wnioski.

- *Funkcja obliczania skrótów jest bardzo ważna.* W przykładzie nasza funkcja przyporządkowała wszystkie klucze do jednego miejsca. Natomiast doskonała funkcja obliczania skrótów odwzorowuje klucze równomiernie w całej tablicy.
- Jeśli listy powiązane staną się bardzo długie, tablica będzie powolna. Jeśli jednak zastosujesz dobrą funkcję obliczania skrótów, długie listy elementów nie powstaną!

Funkcje obliczania skrótów są ważne. Dobra funkcja tego typu zapewni ograniczoną liczbę kolizji. Jak w takim razie wybrać dobrą funkcję obliczania skrótów? Tego dowiesz się w następnym podrozdziale.

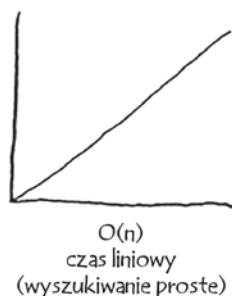
Wydajność

Na początku tego rozdziału przytoczyłem przykład z zieleniakiem. Potrzebowaliśmy rozwiązania, które pozwalałoby natychmiastowo sprawdzać ceny różnych produktów. Jak już wiesz, tablice skrótów są bardzo szybkie.

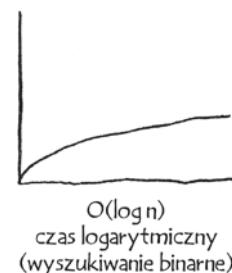
	Średni przypadek	Najgorszy przypadek
Wyszukiwanie	$O(1)$	$O(n)$
Wstawianie	$O(1)$	$O(n)$
Usuwanie	$O(1)$	$O(n)$

Wydajność tablic skrótów

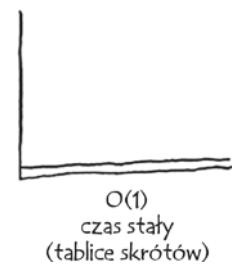
W średnim przypadku w tablicach skrótów wszystkie operacje zajmują $O(1)$. Wartość $O(1)$ nazywana jest **czasem stałym**. Jeszcze o nim nie wspomniałem. Czas stały nie oznacza, że wszystko dzieje się natychmiast, tylko że ilość czasu potrzebnego na wykonanie różnych operacji jest zawsze taka sama niezależnie od rozmiaru tablicy skrótów. Wiesz np., że wyszukiwanie prostego charakteryzuje się czasem liniowym.



Wyszukiwanie binarne jest szybsze, ponieważ charakteryzuje się czasem logarytmicznym.



Natomiast w tablicy skrótów wyszukiwanie elementów trwa zawsze tyle samo czasu.



Widzisz tę poziomą linię? Oznacza ona, że niezależnie od tego, czy tablica zawiera jeden element, czy miliard elementów, znalezienie w niej czegoś zawsze zajmie tyle samo czasu. Z czasem stałym mieliśmy już do czynienia — charakteryzuje się nim pobieranie elementów z tablicy. Nie ma znaczenia, ile elementów zawiera tablica — pobranie z niej jednego elementu zawsze trwa tyle samo czasu. W średnim przypadku tablice skrótów są bardzo szybkie. Natomiast w najgorszym przypadku wszystkie operacje mają czas liniowy, a więc są bardzo powolne. Poniżej znajduje się tabela zawierająca porównanie tablic skrótów z tablicami i listami.

	Tablice skrótów (średni)	Tablice skrótów (najgorszy)	Tablice	Listy powiązane
Wyszukiwanie	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Wstawianie	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Usuwanie	$O(1)$	$O(n)$	$O(n)$	$O(1)$

Zwróć uwagę na wydajność tablic skrótów w średnim przypadku. W wyszukiwaniu (pobieraniu elementu o określonym indeksie) tablice skrótów dorównują zwykłym tablicom i są tak samo szybkie jak listy powiązane we wstawianiu i usuwaniu elementów. Innymi słowy, mają wszystkie zalety obu rozwiązań! Jednak w najgorszym przypadku tablice skrótów są potwornie wolne we wszystkich operacjach. Dlatego należy zadbać, aby w tablicy skrótów nigdy nie dopuścić do najgorszego przypadku. Należy zatem unikać kolizji. A żeby uniknąć kolizji, trzeba zapewnić:

- niski współczynnik zapełnienia,
- dobrą funkcję obliczania skrótów.

Uwaga

Zanim zaczniesz czytać następny punkt, wiedz, że jego lektura nie jest obowiązkowa. Omawiam w nim kwestie dotyczące implementacji tablic skrótów, ale nigdy nie będziesz musiał robić tego samodzielnie. Bez względu na to, jakiego języka programowania używasz, z pewnością ma on wbudowaną gotową implementację tablic skrótów. Możesz z niej skorzystać, mając pewność, że będzie się charakteryzowała doskonałą wydajnością. W następnym punkcie odkrywam nieco tajników tych struktur danych.

Współczynnik zapełnienia

Współczynnik zapełnienia tablicy skrótów można łatwo obliczyć.

Dane w tablicach skrótów są przechowywane w tablicach, więc należy policzyć zajęte komórki w tablicy. Przykładowo współczynnik zapełnienia poniższej tablicy skrótów wynosi $2/5$ czyli $0,4$.

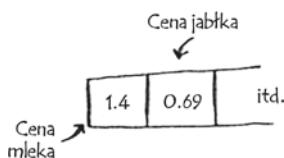


Jaki jest współczynnik zapełnienia tej tablicy skrótów?



Jeśli powiedziałeś 1/3, to dobrze. Współczynnik zapełnienia jest miarą określającą, jaka część tablicy jest jeszcze pusta.

Powiedzmy, że chcemy zapisać ceny 100 produktów i mamy tablicę skrótów zawierającą 100 miejsc. W najlepszym przypadku każdy element zostanie przyporządkowany do osobnej komórki.



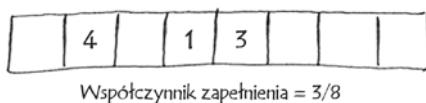
Współczynnik zapełnienia tej tablicy skrótów wynosi 1. A co by się stało, gdyby w tablicy było tylko 50 miejsc? Wówczas współczynnik zapełnienia wynosiłby 2. W takiej sytuacji nie ma szans, aby każdy element miał własną komórkę, ponieważ miejscówek jest po prostu za mało! Współczynnik zapełnienia wyższy od 1 oznacza, że tablica zawiera więcej elementów niż ma miejsc. W takiej sytuacji należy zwiększyć liczbę miejscówek. Operację tę nazywa się **zmianą rozmiaru**. Powiedzmy np., że mamy poniższą tablicę, która — jak widać — wkrótce zostanie zapełniona.



Należy powiększyć tę tablicę skrótów. Najpierw utworzymy nową, większą tablicę. Zwykle tworzy się strukturę dwa razy większą niż poprzednia.



Następnie przy użyciu funkcji `hash` przenosimy wszystkie elementy z poprzedniej tablicy skrótów do nowej.

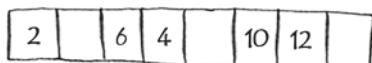


Współczynnik zapełnienia tej nowej tablicy wynosi $3/8$, a więc jest o wiele lepszy! Im niższy współczynnik zapełnienia, tym mniejsze ryzyko występowania kolizji i lepsza wydajność struktury. Dobrą zasadą, której warto przestrzegać, jest zwiększenie tablicy skrótów, gdy jej współczynnik zapełnienia przekroczy 0,7.

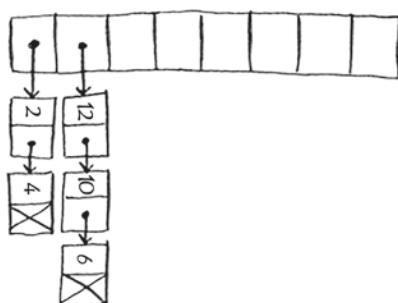
Pewnie sobie myślisz, że to całe powiększanie zajmuje bardzo dużo czasu. I masz rację! Zmiana rozmiaru to bardzo kosztowna operacja i dlatego należy wykonywać ją jak najrzadziej, ale średnio zmiana rozmiaru tablic skrótów zajmuje $O(1)$ czasu.

Dobra funkcja obliczania skrótów

Dobra funkcja obliczania skrótów równomiernie rozprowadza wartości po tablicy.



Słaba funkcja obliczania skrótów zapisuje wartości w grupach i powoduje liczne kolizje.



Co to jest dobra funkcja obliczania skrótów? To jest coś, o co nigdy nie będziesz musiał się martwić — robią to za Ciebie starsze panie i starsi panowie z brodami, którzy siedzą w ciemnych pokojach przed ekranami monitorów. Jeśli naprawdę Cię to interesuje, poszukaj informacji o funkcji SHA (krótki opis znajduje się też w ostatnim rozdziale). Można jej użyć jako funkcji obliczania skrótów.

ĆWICZENIA

Dobra funkcja obliczania skrótów powinna zapewniać równomierną dystrybucję, tzn. powinna jak najszerzej rozmieszczać elementy w strukturze. Najgorsza jest funkcja przypisująca wszystkie elementy do tej samej miejscowości w tablicy skrótów.

Wyobraź sobie, że masz cztery poniższe funkcje obliczania skrótów pobierające łańcuchy.

- A. Funkcja zwracająca 1 dla wszystkich danych wejściowych.
- B. Funkcja wykorzystująca jako indeks długość otrzymanego na wejściu łańcucha.
- C. Funkcja wykorzystująca jako indeks pierwszą literę otrzymanego na wejściu łańcucha, tak że wszystkie napisy zaczynające się na *a* są grupowane w jednej komórce itd.
- D. Funkcja zamieniająca każdą literę na liczbę pierwszą: $a = 2, b = 3, c = 5, d = 7, e = 11$ itd. Dla podanego łańcucha funkcja obliczania skrótów oblicza sumę wszystkich znaków i dzieli ją bez reszty przez rozmiar tablicy skrótów. Jeśli np. rozmiar tablicy skrótów wynosi 10 i zostanie przekazany łańcuch „gad”, to indeks tego łańcucha wyniesie $17 + 2 + 7 \% 10 = 26 \% 10 = 2$.

Która z powyższych funkcji zapewniłaby dobry rozkład w poniższych przypadkach przy założeniu, że rozmiar tablicy skrótów wynosi 10 miejsc.

5.5. Książka telefoniczna, w której kluczami są imiona, a wartościami numery telefonów. Dane są następujące imiona: Eliza, Bernard, Bożydar i Daniel.

5.6. Przypisanie rozmiaru baterii do jej mocy. Dostępne są rozmiary A, AA, AAA oraz AAAA.

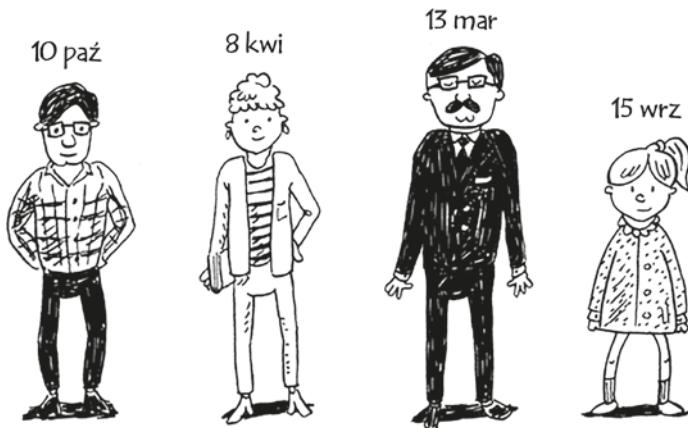
5.7. Przypisanie tytułów książek do autorów. Dane są następujące tytuły: *Maus*, *Fun Home* i *Watchmen*.

Powtórzenie

Prawie na pewno nigdy nie będziesz musiał samodzielnie implementować tablicy skrótów, ponieważ implementacja taka powinna być dostępna w języku programowania, którego będziesz używać. W Pythonie można posługiwać się tablicami skrótów, przyjmując założenie, że uda się osiągnąć średnią wydajność, czyli czas stały.

Tablice skrótów są zaawansowanymi strukturami danych, które charakteryzują się bardzo dużą szybkością działania i umożliwiają modelowanie danych na różne sposoby. Szybko może się okazać, że będziesz ich używać codziennie.

- Tablicę skrótów można utworzyć z połączenia funkcji obliczania skrótów z tablicą.
- Kolizje są złe. Funkcja obliczania skrótów powinna minimalizować ryzyko występowania kolizji.
- Tablice skrótów mają bardzo szybkie operacje wyszukiwania, wstawiania i usuwania.
- Tablice skrótów dobrze nadają się do modelowania relacji między elementami.
- Gdy współczynnik zapełnienia tablicy skrótów osiągnie wartość 0,7, czas zwiększyć rozmiar struktury.
- Tablice skrótów używa się jako pamięci podręcznej (np. na serwerach internetowych).
- Tablice skrótów są świetne w wykrywaniu duplikatów.





W tym rozdziale:

- nauczysz się modelować sieci przy użyciu nowej abstrakcyjnej struktury danych — grafu,
- poznasz technikę wyszukiwania wszerz, czyli algorytm, z pomocą którego można przeszukiwać grafy w celu znalezienia odpowiedzi na pytania w rodzaju: „Jaka jest najkrótsza droga do X?”,
- dowiesz się, jaka jest różnica między grafami skierowanymi i nieskierowanymi,
- poznasz sortowanie topologiczne, czyli nowy rodzaj algorytmu sortowania, który ujawnia zależności między węzłami.

Ten rozdział stanowi wprowadzenie do grafów. Najpierw wyjaśniam, czym w ogóle są grafy (nie mają osi X ani Y), po czym przechodzę do omówienia pierwszego algorytmu działającego na tych strukturach o nazwie **wyszukiwanie wszerz** (ang. *breadth-first search* — BFS).

Wyszukiwanie wszerz umożliwia znalezienie najkrótszej drogi między dwoma punktami. Jednak pojęcie najkrótszej drogi można definiować na wiele sposobów! Za pomocą wyszukiwania wszerz można:

- utworzyć sztuczną inteligencję do gry w szachy, która będzie obliczać najmniejszą liczbę ruchów do zwycięstwa;

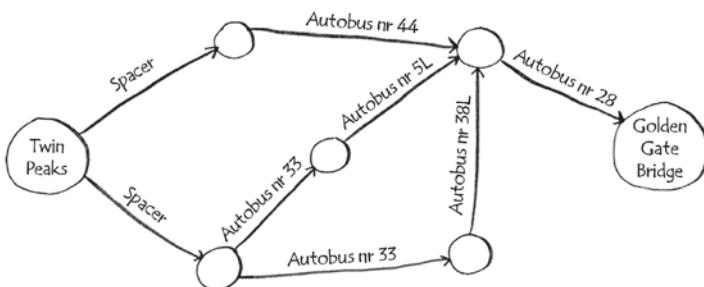
- napisać moduł sprawdzania pisowni (najmniejsza liczba zmian prowadząca do poprawienia źle napisanego słowa — np. czytnil -> czytnik to jedna poprawka);
- znaleźć najbliższego lekarza.

Algorytmy grafowe należą do najbardziej przydatnych algorytmów, jakie znam. Dlatego następne rozdziały przestudiuj wyjątkowo dokładnie, ponieważ z algorytmów tych będziesz wielokrotnie korzystać.

Wprowadzenie do grafów

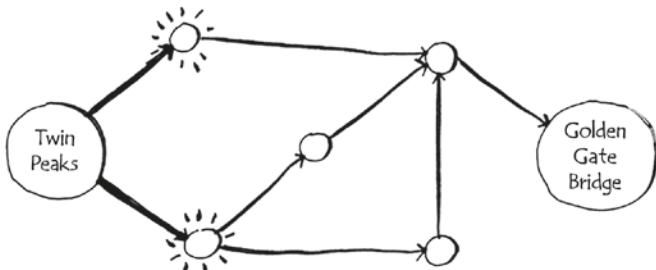


Wyobraź sobie, że jesteś w San Francisco i chcesz dostać się z Twin Peaks na Golden Gate Bridge. Zamierzasz pojechać autobusem, robiąc po drodze jak najmniej przesiadek. Oto dostępne możliwości.

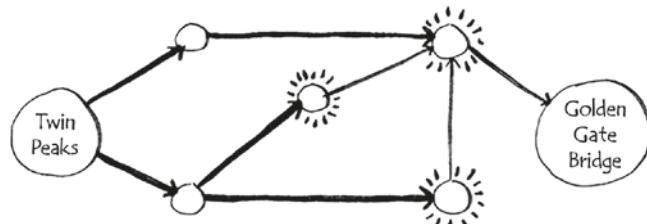


Jakiego algorytmu użyjesz, aby znaleźć drogę z najmniejszą liczbą zmian?

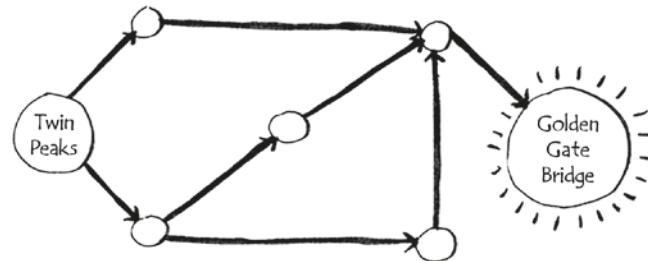
Czy da się dotrzeć na miejsce jednym środkiem? Oto wszystkie miejsca, do których można dotrzeć w jednym etapie.



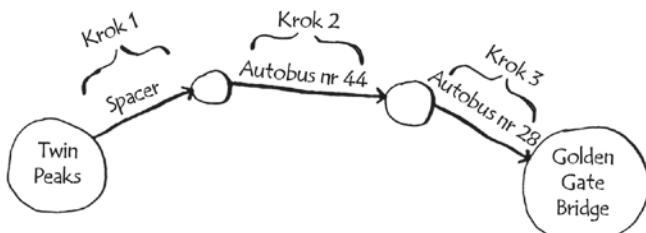
Most nie jest zaznaczony, więc nie da się do niego dotrzeć w jednym kroku.
Czy wystarczą dwa kroki?



Nadal nie dotarliśmy do celu, a więc nie da się dotrzeć do mostu w dwóch krokach. Może w takim razie trzy kroki?



Bingo! Pojawił się Golden Gate Bridge. Zatem trasa do niego z Twin Peaks jest trzyetapowa, co pokazano na poniższym rysunku.



Są jeszcze inne drogi, z których można skorzystać, ale każda z nich jest dłuższa (cztery kroki) od wybranej. Za pomocą algorytmu sprawdziłem, że najkrótsza trasa do mostu składa się z trzech etapów. Tego rodzaju zadania nazywają się **problemami wyboru najkrótszej drogi** (ang. *shortest-path problem*). Zawsze próbujemy znaleźć najkrótszą drogę do czegoś, np. do domu znajomego albo do wygranej w partii szachów. Algorytm rozwiązujejący problem najkrótszej drogi nazywa się **wyszukiwaniem wszerz**.

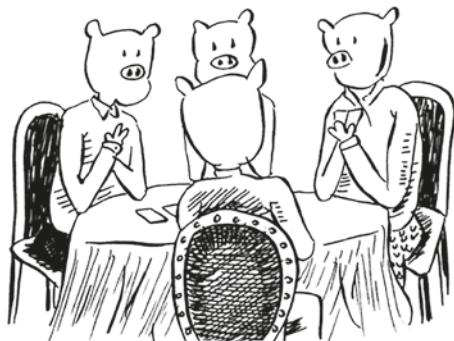
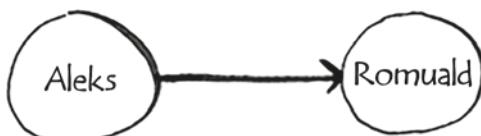
Aby dowiedzieć się, jak dotrzeć z Twin Peaks do Golden Gate Bridge, należy wykonać dwie czynności:

1. stworzyć model problemu w postaci grafu,
2. rozwiązać problem za pomocą wyszukiwania wszerz.

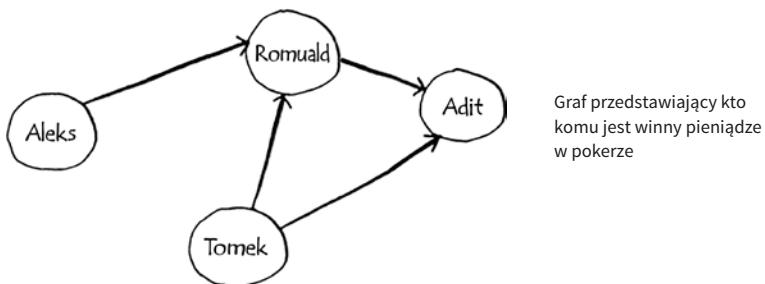
W następnym podrozdziale najpierw piszę, czym są grafy, a później bardziej szczegółowo omawiam technikę wyszukiwania wszerz.

Czym jest graf

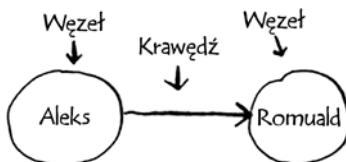
Graf jest modelem zbioru połączeń. Wyobraź sobie np., że grasz ze znajomymi w pokera i za pomocą modelu chcesz przedstawić, kto komu jest winien pieniędze. Możesz więc stwierdzić: „Aleks jest winny Romualdowi pieniądze”.



Kompletny graf mógłby wyglądać tak, jak na poniższym rysunku.



Aleks jest winny pieniądze Romualdowi, Tomek jest winny Aditowi itd. Każdy graf składa się z **węzłów** i **krawędzi**.



To w zasadzie wszystko na ten temat! Grafy składają się z węzłów i krawędzi. Węzeł może być bezpośrednio połączony z wieloma innymi węzłami, które nazywają się jego **sąsiadami**. Na powyższym grafie Romuald jest sąsiadem Alekса. Adit nie jest sąsiadem Alekса, ponieważ nie ma między nimi bezpośredniego połączenia. Natomiast Adit jest sąsiadem Romualda i Tomka.

Grafy służą do modelowania połączeń między różnymi przedmiotami. Wiedząc to, możemy przejść do wyszukiwania wszerz.

Wyszukiwanie wszerz

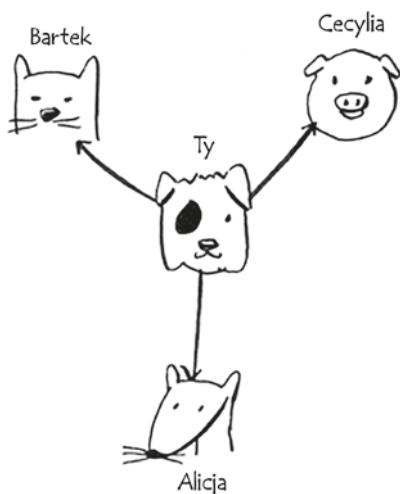
W rozdziale 1. przestudiowałeś już algorytm wyszukiwania zwany wyszukiwaniem binarnym. Wyszukiwanie wszerz to także algorytm tego typu, ale przeznaczony do pracy z grafami. Za jego pomocą można znaleźć odpowiedzi na dwa rodzaje pytań.

- Typ 1.: czy istnieje droga z węzła A do węzła B?
- Typ 2.: jaka jest najkrótsza droga z węzła A do węzła B?

Z wyszukiwaniem wszerz spotkaliśmy się już przy okazji obliczania najkrótszej drogi z Twin Peaks do Golden Gate Bridge. Wówczas szukaliśmy odpowiedzi na pytanie drugiego typu, czyli o najkrótszą drogę. Teraz dokładniej przyjrzymy się temu algorytmowi. Zadamy sobie pytanie pierwszego typu: „Czy istnieje droga?”.



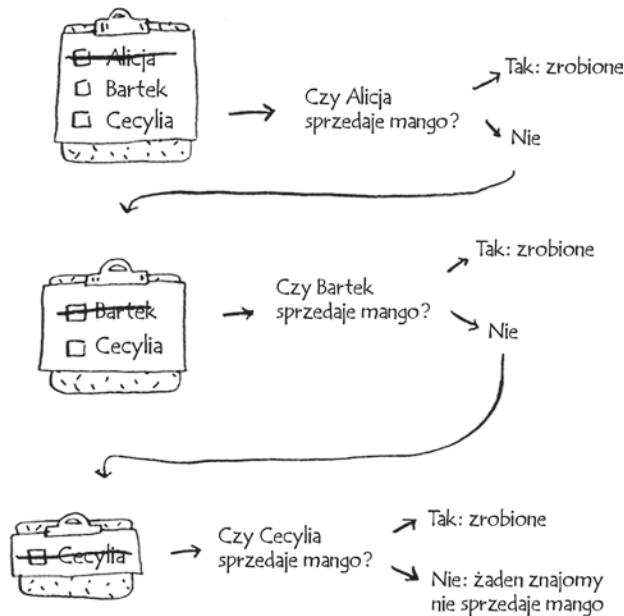
Wyobraź sobie, że jesteś dumnym posiadaczem sadu mangowego. Szukasz handlarza owocami mango, który zechciałby sprzedawać Twoje owoce. Jesteś połączony z jakimś sprzedawcą tych owoców na Facebooku? Możesz poszukać wśród swoich znajomych.



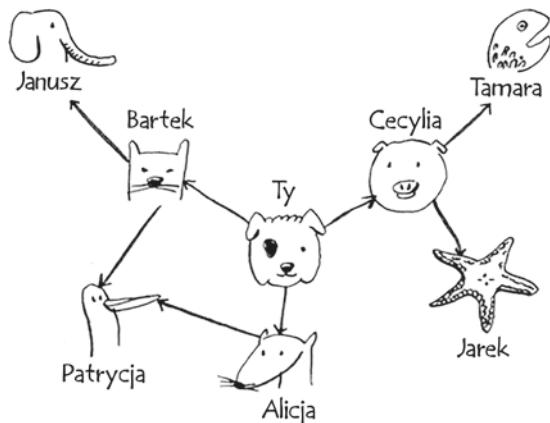
Taki sposób wyszukiwania jest bardzo prosty. Najpierw należy przygotować listę znajomych do przeszukania.



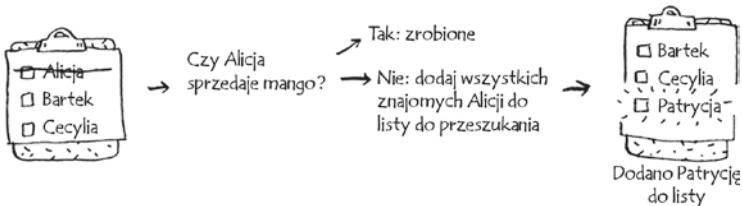
Teraz oglądamy profil każdej osoby na liście i sprawdzamy, czy sprzedaje mango.



Powiedzmy, że żaden z naszych znajomych nie sprzedaje mango i musimy przeszukać także znajomych znajomych.



Za każdym razem, gdy sprawdzasz kogoś z listy, dodajesz wszystkich jego znajomych do listy.



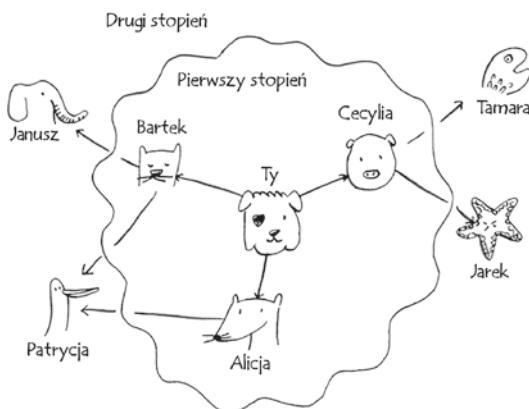
W ten sposób przeszukasz nie tylko swoich znajomych, ale też i ich znajomych. Pamiętaj, że szukasz w swojej sieci sprzedawcy owoców mango. Jeśli więc Alicja nie zajmuje się tego typu działalnością, dodajesz do listy także jej znajomych. Znaczy to, że sprawdzisz wszystkich jej znajomych, a potem ich znajomych itd. Za pomocą tego algorytmu przeszukasz całą sieć, aż w końcu natrafisz na sprzedawcę owoców mango. Tak właśnie działa algorytm wyszukiwania wszerz.

Szukanie najkrótszej drogi

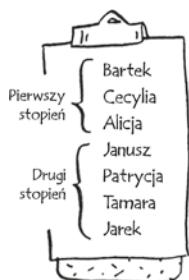
Dla przypomnienia, poniżej jeszcze raz przedstawiam dwa rodzaje pytań, na jakie można znaleźć odpowiedź za pomocą algorytmu wyszukiwania wszerz.

- Typ 1.: czy istnieje droga z węzła A do węzła B? (Czy w mojej sieci znajduje się jakiś sprzedawca owoców mango)?
- Typ 2.: jaka jest najkrótsza droga z węzła A do węzła B? (Kto jest najbliższym sprzedawcą owoców mango)?

Pokazałem już, jak znaleźć odpowiedź na pytanie 1. Teraz poszukamy odpowiedzi na pytanie 2. Potrafisz znaleźć najbliższego sprzedawcę owoców mango? Przykładowo Twoi znajomi są Twoimi połączonymi pierwszego stopnia, a ich znajomi — drugiego stopnia.



Wolałbyś połączenie pierwszego niż drugiego stopnia, a połączenie drugiego stopnia od trzeciego stopnia itd. Dlatego nie powinieneś przeszukiwać żadnych połączeń na drugim poziomie, dopóki nie przejrzyz wszystkich na pierwszym. I tak właśnie działa wyszukiwanie wszerz! Algorytm rozchodzi się na boki od punktu startowego i w ten sposób najpierw przeszukuje wszystkie połączenia pierwszego stopnia i dopiero po ich dokładnym sprawdzeniu przechodzi na drugi poziom. A teraz krótka zagadka: kto zostanie sprawdzony jako pierwszy — Cecylia czy Janusz? Oto odpowiedź: Cecylia jest połączeniem pierwszego stopnia, a Janusz — drugiego, dlatego najpierw algorytm sprawdzi Cecylię.



Można też spojrzeć na to z innej perspektywy. Połączenia pierwszego stopnia są dodawane do listy elementów do przeszukania przed połączaniami drugiego stopnia.

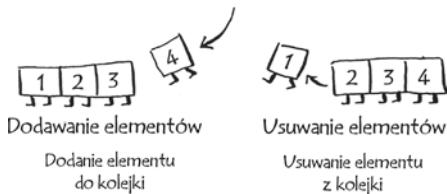
Po kolej przeglądamy elementy na liście i sprawdzamy, czy reprezentowane przez nie osoby sprzedają mango. Połączenia pierwszego stopnia zostaną przeszukane przed połączaniami drugiego stopnia, dzięki czemu wkrótce znajdziemy najbliższego sprzedawcę. Wyszukiwanie wszerz pozwala nie tylko znaleźć drogę z punktu A do B, ale dodatkowo umożliwia znalezienie najkrótszej drogi.

Zauważ, że metoda ta działa tylko w przypadku przeszukiwania kontaktów w kolejności ich dodawania do listy. Jeśli zatem Cecylię dodano do listy przed Januszem, to Cecylia musi zostać sprawdzona jako pierwsza. A co się stanie, jeśli sprawdzimy Janusza przed Cecylią i oboje będą sprzedawcami mango? Janusz jest przecież kontaktem drugiego stopnia, a Cecylia — pierwszego. Wówczas znajdziemy sprzedawcę mango, który nie jest nam najbliższy w naszej sieci. Dlatego osoby należy sprawdzać w takiej kolejności, w jakiej zostały dodane do listy. Istnieje nawet specjalna struktura do takiego przechowywania danych, jest to **kolejka**.

Kolejki

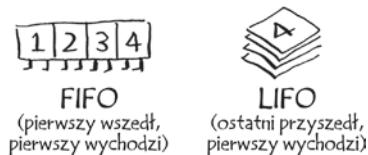
Kolejka w programowaniu działa dokładnie tak samo jak w realnym świecie. Wyobraź sobie, że razem ze znajomym stajecie w kolejce do autobusu. Jeśli стоisz przed kolegą, do autobusu wejdzieš pierwšzy. Tak samo działa struktura danych zwana kolejką i można ją porównać do stosu. W kolejce nie ma swobodnego dostępu do elementów i można wykonywać tylko dwie operacje, czyli *dodawanie* i *usuwanie* elementów.





Jeśli dodasz do kolejki dwa elementy, to ten, który został dodany wcześniej, zostanie zdjęty przed drugim. Strukturę tę możesz więc wykorzystać do przeszukiwania swojej listy! Osoby dodane do listy wcześniej zostaną z niej pobrane i sprawdzone wcześniej.

Kolejka jest tzw. strukturą danych typu **FIFO** (ang. *First In, First Out* — pierwszy wszedł, pierwszy wychodzi). Dla porównania stos jest strukturą typu **LIFO** (ang. *Last In, First Out* — ostatni przyszedł, pierwszy wychodzi).

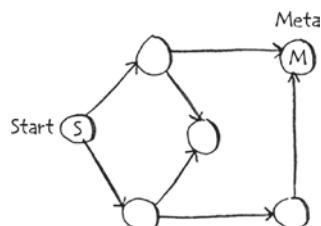


Wiedząc, jak działa kolejka, możesz zaimplementować algorytm wyszukiwania wszerz!

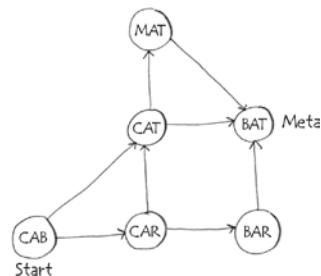
ĆWICZENIA

Przeprowadź wyszukiwanie wszerz każdego z poniższych grafów, aby znaleźć rozwiązanie.

- 6.1.** Znajdź długość najkrótszej drogi od linii startu do mety.



- 6.2.** Znajdź długość najkrótszej drogi z „cab” do „bat”.

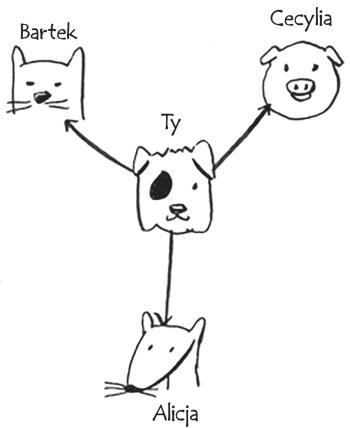
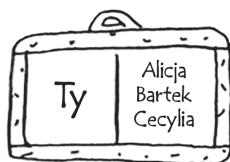


Implementacja grafu

Najpierw musimy napisać implementację grafu w postaci kodu źródłowego. Graf składa się z kilku węzłów.

Każdy węzeł jest połączony z węzłami sąsiednimi. Jak wyrazić relację typu „Ty -> Bartek”? Na szczęście znamy strukturę danych, za pomocą której bardzo łatwo modeluje się takie relacje, to **tablica skrótów!**

Przypomnij, że w tablicy skrótów można odwzorować klucze jako wartości. W tym przypadku chcemy powiązać węzeł z wszystkimi jego sąsiadami.

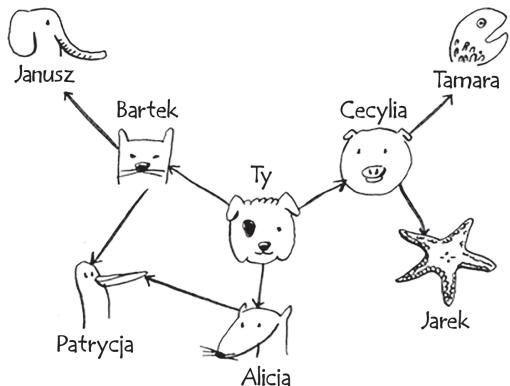


W Pythonie można to zapisać tak.

```
graph = {}
graph["ty"] = ["alicja", "bartek", "cecylia"]
```

Zwróć uwagę, że klucz "ty" jest powiązany z tablicą. Dzięki temu instrukcja `graph["ty"]`wróci tablicę wszystkich sąsiadów tego elementu.

Graf jest tylko zbiorem węzłów i krawędzi, a to wystarczy do utworzenia tej struktury w Pythonie. A gdybyśmy chcieli zaimplementować większy graf, taki jak poniższy?



Oto odpowiedni kod w Pythonie.

```
graph = []
graph["ty"] = ["alicja", "bartek", "cecylia"]
graph["bartek"] = ["janusz", "patrycja"]
graph["alicja"] = ["patrycja"]
graph["cecylia"] = ["tamara", "jarek"]
graph["janusz"] = []
graph["patrycja"] = []
graph["tamara"] = []
graph["jarek"] = []
```

Czy kolejność dodawania par klucz-wartość ma znaczenie? Czy jakieś znaczenie ma to, że napiszemy:

```
graph["cecylia"] = ["tamara", "jarek"]
graph["janusz"] = []
```

zamiast:

```
graph["janusz"] = []
graph["cecylia"] = ["tamara", "jarek"]
```

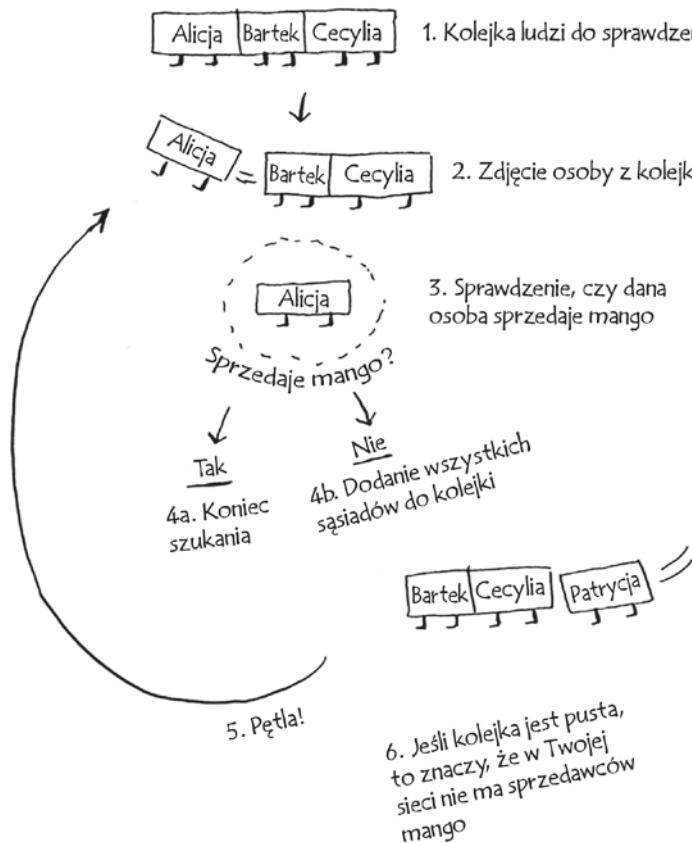
Przypomnij sobie poprzedni rozdział. Odpowiedź brzmi: to nie ma znaczenia. Tablice skrótów nie są uporządkowane, więc kolejność dodawania do nich par klucz-wartość nie ma znaczenia.

Janusz, Patrycja, Tamara i Jarek nie mają żadnych sąsiadów. Są strzałki wskazujące w ich kierunku, ale od nich nie wychodzą żadne strzałki. Grafy, w których występują tylko takie jednokierunkowe relacje, nazywają się **grafami skierowanymi**. Zatem Janusz jest sąsiadem Bartka, ale Bartek nie jest sąsiadem Janusza. Graf nieskierowany nie zawiera żadnych strzałek i węzły w każdej parze są swoimi sąsiadami nawzajem. Przykładowo poniższe dwa grafy są równoważne.



Implementacja algorytmu

Dla przypomnienia poniżej przedstawiam, jak ma działać implementacja.



Utworzmy sekwencję elementów na początek. W Pythonie do tego celu można użyć funkcji deque (kolejka dwustronna).

```
from collections import deque
search_queue = deque() Utworzenie nowej kolejki.
search_queue += graph["ty"] Dodanie wszystkich sąsiadów do kolejki.
```

Przypomnę, że instrukcja `graph["ty"]` zwróci listę wszystkich sąsiadów określonego elementu, np. `["alicja", "bartek", "cecylia"]`. Wszystkie te elementy zostaną dodane do naszej wyszukiwarki.



Zobaczmy resztę.

```

while search_queue: ←..... Dopóki kolejka nie jest pusta...
    person = search_queue.popleft() ←..... Pobranie pierwszego elementu z kolejki.
    if person_is_seller(person): ←..... Sprawdzenie, czy dana osoba sprzedaje mango.
        print person + " sprzedaje mango!" ←..... Tak, ta osoba sprzedaje mango.
        return True
    else:
        search_queue += graph[person] ←..... Nie sprzedaje mango. Dodanie wszystkich
                                         znajomych tej osoby do kolejki do przeszukania.
return False ←..... Jeśli dotarliśmy tutaj, znaczy to, że nikt
                  w kolejce nie jest sprzedawcą mango.

```

I jeszcze jedna rzecz — nadal potrzebujemy funkcji `person_is_seller`, która będzie nas informować, czy dana osoba jest sprzedawcą mango. Oto definicja tej funkcji.

```
def person_is_seller(name):
    return name[-1] == 'm'
```

Funkcja sprawdza, czy imię osoby kończy się literą m. Jeśli tak, znaczy to, że ta osoba sprzedaje mango. To oczywiście bardzo niemądre rozwiązańe, ale na potrzeby tego przykładu wystarczy. Teraz zobaczymy wyszukiwanie wszerz w akcji.



itd.

Algorytm będzie działał w ten sposób, aż:

- znajdzie sprzedawcę mango,
- opróżni kolejkę i stwierdzi, że nie było w niej sprzedawcy mango.

Alicja i Bartek mają wspólną znajomą — Patrycję. W efekcie Patrycja zostanie dodana do kolejki dwa razy — raz przy dodawaniu znajomych Alicji i drugi raz przy dodawaniu znajomych Bartka. Patrycję będziemy więc mieli zapisaną dwa razy.

Aby jednak dowiedzieć się, czy Patrycja sprzedaje mango, wystarczy ją sprawdzić tylko raz. Jeśli zrobimy to dwa razy, tylko zmarnujemy trochę czasu na wykonywanie niepotrzebnych zadań. Dlatego każdą sprawdzaną osobę należy oznaczyć, aby ewentualnie niepotrzebnie nie sprawdzać jej ponownie.

Zaniechanie tego może nawet przyczynić się do powstania pętli nieskończonej. Powiedzmy, że nasz graf wygląda tak, jak na poniższym rysunku.



Na początku kolejka do przeszukania zawiera wszystkich Twoich sąsiadów.



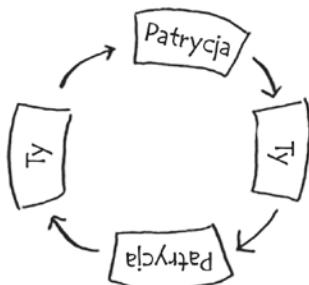
Sprawdzamy więc Patrycję. Nie jest handlarzem owocami mango, więc do kolejki dodajemy wszystkich jej sąsiadów.



Następnie sprawdzamy siebie. Stwierdzamy, że nie sprzedajemy owoców mango, więc dodajemy do kolejki wszystkich swoich sąsiadów.



I tak dalej. Powstaje nieskończona pętla, ponieważ w kolejce będą na przemian zapisywane dwa elementy — ty i patrycja.



Przed sprawdzeniem jakiejkolwiek osoby zawsze należy najpierw zweryfikować, czy osoba ta nie została już wcześniej sprawdzona. W tym celu można utworzyć listę sprawdzonych osób.

Poniżej znajduje się ostateczna wersja algorytmu wyszukiwania wszerz z uwzględnieniem ostatnich uwag.

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = [] Przy użyciu tej tablicy kontrolujemy,
które osoby już sprawdziliśmy.
    while search_queue:
        person = search_queue.popleft()
        if not person in searched: Sprawdzamy osobę tylko wtedy,
gdy wcześniej jej nie sprawdzaliśmy.
            if person_is_seller(person):
                print person + " sprzedaje mango!"
                return True
            else:
                search_queue += graph[person]
                searched.append(person) Oznacza osobę jako sprawdzoną.
    return False

search("ty")
```

Uruchom ten kod, aby go wypróbować. Możesz zmienić funkcję `person_is_seller` na coś ciekawszego, aby sprawdzić, czy wydrukuje oczekiwany wynik.

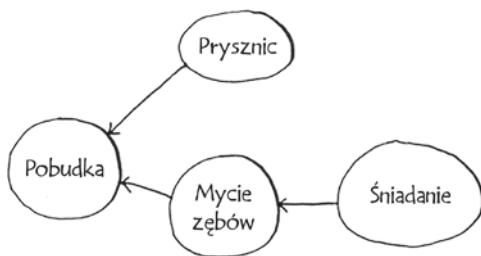
Czas wykonywania

Jeśli okaże się, że przeszukałeś całą swoją sieć, będzie to znaczyło, że przemierzyłeś każdą krawędź (przypomnij, że krawędź to strzałka lub połączenie między dwiema osobami). W związku z tym czas wykonywania opisywanego algorytmu wynosi przynajmniej $O(\text{liczba krawędzi})$.

Dodatkowo algorytm tworzy kolejkę osób do przeszukania. Czas dodawania jednej osoby do tej kolejki jest stały: $O(1)$. Wykonanie tej operacji dla każdej osoby zajmie w sumie $O(\text{liczba ludzi})$ czasu. Wyszukiwanie wszerz zajmuje więc $O(\text{liczba ludzi} + \text{liczba krawędzi})$ czasu, co najczęściej zapisuje się w formie $O(V + E)$, gdzie V oznacza liczbę wierzchołków, a E liczbę krawędzi.

ĆWICZENIE

Oto niewielki graf przedstawiający mój typowy poranek.



Z tego grafu wynika, że nie mogę zjeść śniadania, dopóki nie wyszczotuję zębów. W związku z tym „śniadanie” jest zależne od „mycia zębów”.

A przecież prysznic nie zależy od mycia zębów, ponieważ mogę go wziąć, zanim przystąpię do higieny jamy ustnej. Na podstawie tego grafu można utworzyć listę przedstawiającą kolejność wykonywania przez mnie porannych czynności:

1. pobudka,
2. prysznic,
3. mycie zębów,
4. śniadanie.

Miejsce prysznica można zmieniać, więc poniższa lista również jest poprawna.

1. pobudka,
 2. mycie zębów,
 3. prysznic,
 4. śniadanie.

6.3. Przyjrzyj się trzem poniższym listom i powiedz, które z nich są poprawne.

A.

1. Pobudka
 2. Prysznic
 3. Śniadanie
 4. Mycie zębów

B.

1. Pobudka
 2. Mycie zębów
 3. Śniadanie
 4. Prysznic

C.

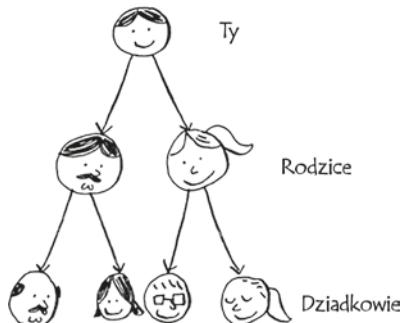
1. Prysznic
 2. Pobudka
 3. Mycie zębów
 4. Śniadanie

6.4. Oto nieco większy graf. Utwórz z niego poprawną listę.

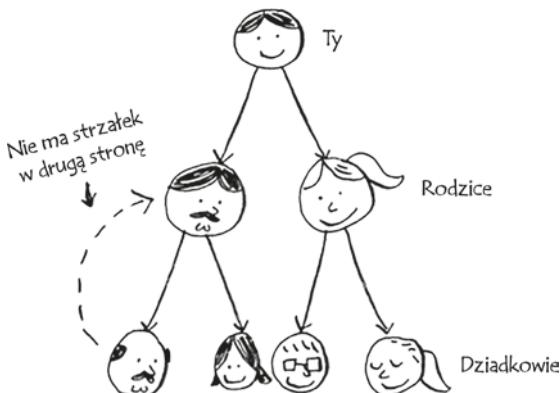


Można powiedzieć, że lista ta jest w pewien sposób posortowana. Jeśli zadanie A zależy od B, to zadanie A występuje na dalszym miejscu w liście. Takie uporządkowanie nazywa się **sortowaniem topologicznym** i jest jednym ze sposobów na sporządzenie listy z grafu. Wyobraź sobie, że planujesz ślub i masz duży graf obejmujący różne zadania do wykonania — i nie wiesz od czego zacząć. W takim przypadku możesz **posortować graf topologicznie**, aby otrzymać listę czynności do wykonania po kolejci.

Spójrz na poniższe drzewo rodzinne.



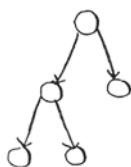
To jest graf, ponieważ zawiera węzły (ludzie) i krawędzie. Każda krawędź wskazuje rodzica danego węzła. Wszystkie krawędzie prowadzą w dół — drzewo rodzinne z krawędziami skierowanymi w górę byłoby bez sensu! Przecież Twój ojciec nie może być ojcem Twojego dziadka!



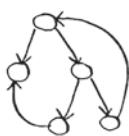
Taką strukturę nazywamy **drzewem**. Jest to specjalny rodzaj grafu, w którym wszystkie krawędzie wskazują w jedną stronę.

6.5. Które z poniższych grafów są także drzewami?

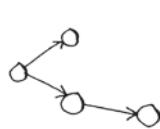
A.



B.

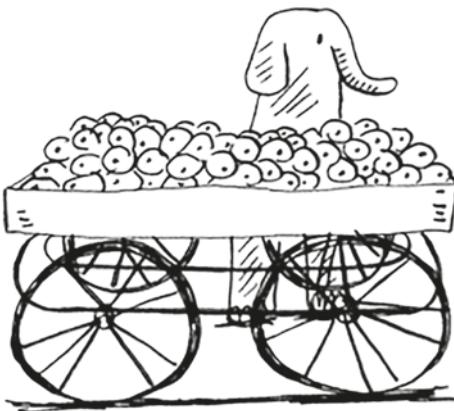


C.



Powtórzenie

- Wyszukiwanie wszerz pozwala stwierdzić, czy istnieje droga z punktu A do B.
- Jeśli istnieją drogi, wyszukiwanie wszerz znajdzie najkrótszą.
- Jeśli masz do rozwiązania zadanie typu: „Znajdź najkrótszy X”, spróbuj utworzyć model problemu w postaci grafu i zastosować wyszukiwanie wszerz.
- Graf skierowany zawiera strzałki i relacje są zgodne z kierunkiem wskazywanym przez te strzałki (rama -> adit oznacza „rama jest winna aditowi pieniądze”).
- Grafy nieskierowane nie mają strzałek i relacje są dwustronne (roman - roksana oznacza „roman umówił się z roksaną, a roksana umówiła się z romanem”).
- Kolejki to struktury typu FIFO (pierwszy na wejściu, pierwszy na wyjściu).
- Stosy są strukturami typu LIFO (ostatni na wejściu, pierwszy na wyjściu).
- Ludzi należy sprawdzać w kolejności ich dodawania do listy, w związku z czym lista ta musi być kolejką. W przeciwnym razie nie otrzymasz najkrótszej drogi.
- Po sprawdzeniu osoby należy zadbać o to, by przez przypadek nie sprawdzić jej jeszcze raz. Niedopilnowanie tego może spowodować powstanie pętli nieskończonej.

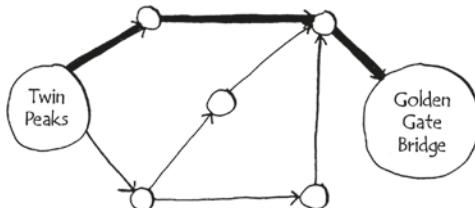




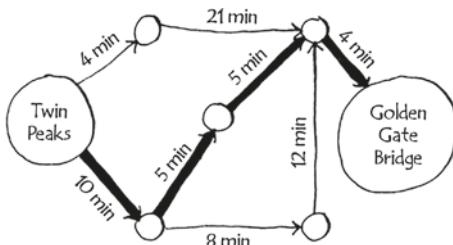
W tym rozdziale:

- będziesz kontynuować odkrywanie grafów i poznasz grafy ważone, tzn. takie, w jakich niektórym krawędziom można przypisywać większą lub mniejszą wagę,
- poznasz algorytm Dijkstry, za pomocą którego można znajdować odpowiedzi na pytania typu: „Jaka jest najkrótsza droga do X?” w odniesieniu do grafów ważonych,
- dowiesz się, czym są cykle w grafach, w których nie działa algorytm Dijkstry.

W poprzednim rozdziale znaleźliśmy drogę z punktu A do punktu B.



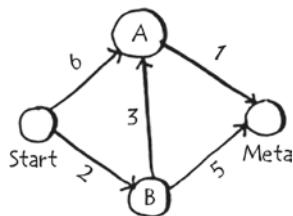
Jednak niekoniecznie jest to droga najszybsza. Wiadomo, że jest najkrótsza, ponieważ zawiera najmniejszą liczbę segmentów (trzy), ale wyobraź sobie, że na każdym odcinku określamy czas potrzebny do jego przebycia. Teraz wyraźnie widać, że jest szybsza trasa.



W poprzednim rozdziale korzystaliśmy z algorytmu wyszukiwania wszerz. Za jego pomocą można znaleźć drogę składającą się z najmniejszej liczby odcinków (pierwszy pokazany tu graf). A co zrobić, aby zamiast tego znaleźć najszybszą trasę (drugi graf)? W tym celu najlepiej skorzystać z **algorytmu Dijkstry**.

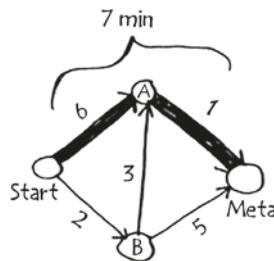
Posługiwanie się algorytmem Dijkstry

Zobaczmy, jak interesujący nas algorytm będzie działał na poniższym grafie.



Każdy odcinek ma określony czas przejazdu w minutach. Za pomocą algorytmu Dijkstry przejdziemy od początku do mety w najkrótszym możliwym czasie.

Gdybyśmy graf ten przekazali do algorytmu wyszukiwania wszerz, otrzymalibyśmy najkrótszą drogę.



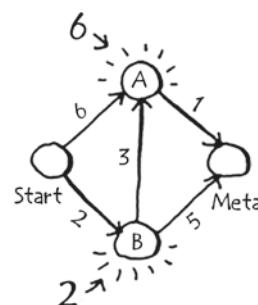
Przebycie tej drogi zajmuje 7 minut. Sprawdzimy, czy istnieje trasa, której pokonanie jest mniej czasochłonne! Wykonywanie algorytmu Dijkstry przebiega w czterech etapach.

1. Najpierw algorytm szuka „najtańszego” węzła, czyli takiego, do którego można dotrzeć w najkrótszym czasie.
2. Następnie aktualizowane są „koszty” węzłów sąsiednich. Za chwilę wyjaśnię, co dokładnie mam na myśli.
3. Czynności są powtarzane dla wszystkich węzłów w grafie.
4. Zostaje obliczona ostateczna droga.

Krok 1.: znalezienie najtańszego węzła. Stojmy na linii startu i zastanawiamy się, czy pójść do węzła A, czy do B. Ile czasu zajmuje dotarcie do każdego z nich?

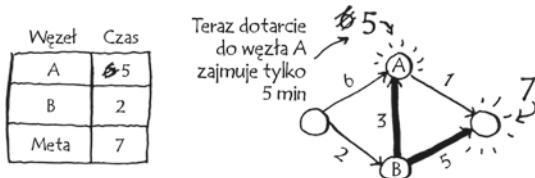
Podróż do węzła A trwa 6 minut, a do węzła B — 2 minuty. Czasu potrzebnego na dotarcie do pozostałych węzłów jeszcze nie znamy.

Ponieważ jeszcze nie wiemy, ile czasu zajmie dotarcie do mety, wstawiamy znak nieskończoności (zaraz się dowiesz dlaczego). Węzeł B jest najbliższym, ponieważ znajduje się dwie minuty drogi od nas.

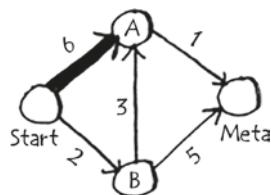


Węzeł	Czas podróży
A	6
B	2
Meta	∞

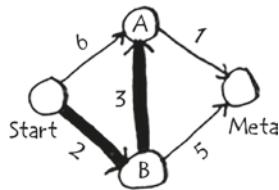
Krok 2.: obliczenie czasu potrzebnego na dotarcie do wszystkich sąsiadów węzła B przez podżeganie krawędzią z węzła B.



Hej, właśnie znaleźliśmy krótszą drogę do węzła A! Wcześniej myśleliśmy, że podróż do węzła A musi trwać sześć minut.



Jeśli pójdziemy przez węzeł B, wystarczy nam tylko pięć minut!



Gdy znajdziesz krótszą drogę dla sąsiada węzła B, zaktualizuj jego koszt. W tym przypadku odkryliśmy:

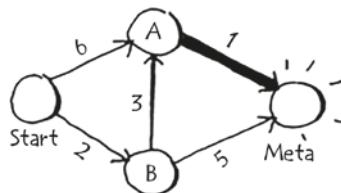
- krótszą drogę do węzła A (pięć zamiast sześciu minut),
- krótszą drogę do mety (siedem minut zamiast nieskończoności).

Krok 3.: powtórzenie czynności!

Krok 1. jeszcze raz: znalezienie węzła, do którego można dotrzeć w najkrótszym czasie. Węzeł B już sprawdziliśmy, więc następny w kolejności do najkrótszego czasu jest węzeł A.

Węzeł	Czas
A	5
B	2
Meta	7

Krok 2. jeszcze raz: aktualizacja kosztów dla sąsiadów węzła A.



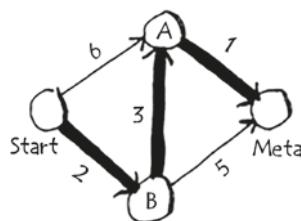
Wow, teraz dotarcie do mety zajmuje tylko sześć minut!

Wykonaliśmy algorytm Dijkstry dla każdego węzła (węzeł mety możemy pominąć). W tym momencie wiemy już, że:

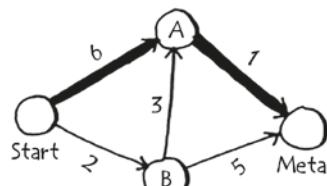
- do węzła B można dotrzeć w dwie minuty,
- do węzła A można dotrzeć w pięć minut,
- do mety można dotrzeć w sześć minut.

Węzeł	Czas
A	5
B	2
Meta	6

Ostatni krok, czyli obliczenie ostatecznej drogi, wykonam dalej w tym rozdziale. Na razie chcę tylko pokazać Ci, jak ta droga będzie wyglądać.

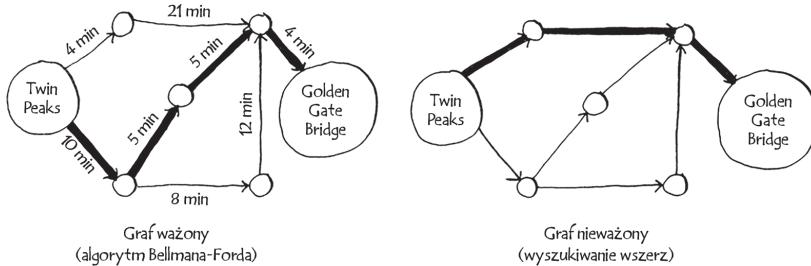


Algorytm wyszukiwania wszerz nie określiłby tej drogi jako najkrótszej, ponieważ składa się ona z trzech odcinków, a z linii startowej do mety można dotrzeć po dwóch odcinkach.



Najkrótsza droga
według wyszukiwania wszerz

W poprzednim rozdziale szukaliśmy najkrótszej drogi między dwoma punktami za pomocą algorytmu wyszukiwania wszerz. Wówczas „najkrótszą drogę” definiowaliśmy jako trasę złożoną z najmniejszej liczby odcinków. Natomiast w algorytmie Dijkstry każdemu odcinkowi przypisuje się liczby, czyli wagę. Dzięki temu algorytm może znaleźć drogę o najmniejszej sumarycznej wadze.



Dla przypomnienia, algorytm Dijkstry składa się z czterech etapów.

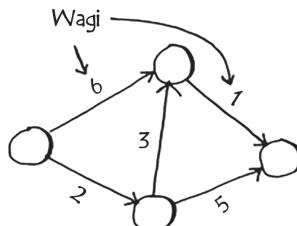
1. Najpierw algorytm szuka „najtańszego” węzła, czyli takiego, do którego można dotrzeć w najkrótszym czasie.
2. Następnie aktualizowane są „koszty” węzłów sąsiednich.
3. Czynności są powtarzane dla wszystkich węzłów w grafie.
4. Zostaje obliczona ostateczna droga (opis metody w następnym podrozdziale!).

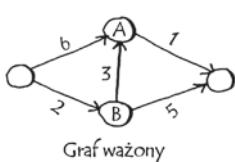
Terminologia

Chciałbym pokazać teraz kilka dodatkowych przykładów działania algorytmu Dijkstry. Wcześniej jednak trzeba nieco uporządkować terminologię.

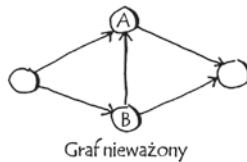
Podczas pracy z algorytmem Dijkstry każda krawędź w grafie ma przypisaną pewną liczbę. Są to tzw. **wagi**.

Graf z określonymi wagami nazywa się **grafem ważonym**, natomiast graf bez określonych wag to **graf nieważony**.



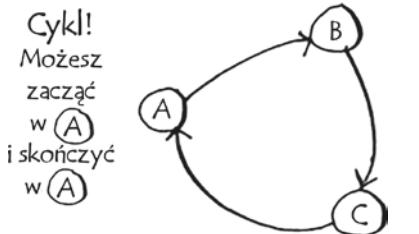


Graf ważony

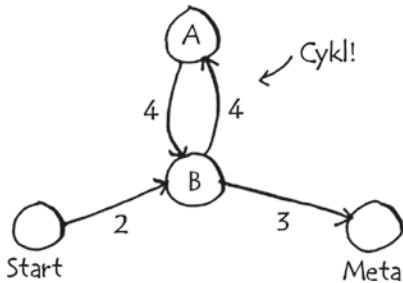


Graf nieważony

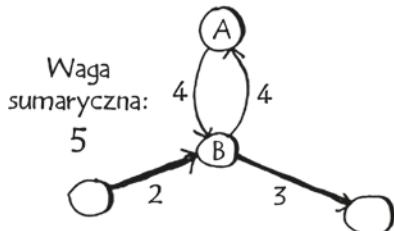
Aby obliczyć najkrótszą drogę w grafie nieważonym, należy użyć algorytmu **wyszukiwania wszerz**. Aby natomiast obliczyć najkrótszą drogę w grafie ważonym, należy skorzystać z **algorytmu Dijkstry**. Grafy mogą też zawierać **cykle**. Cykl wygląda tak.



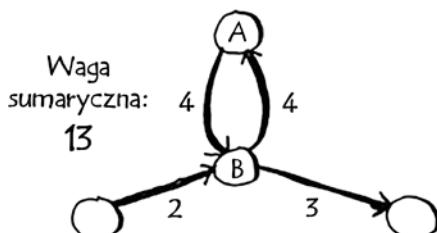
Znaczy to, że podróż można zacząć w węźle A, przejść graf dookoła i skończyć z powrotem w tym samym węźle, w którym się zaczęło. Powiedzmy, że chcemy znaleźć najkrótszą drogę w tym grafie, który zawiera cykl.



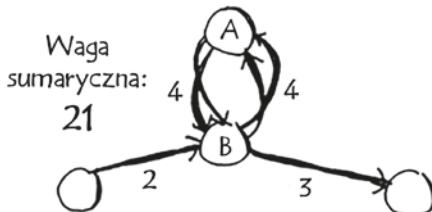
Czy przechodzenie przez cykl ma sens? W grafie jest ścieżka, która pozwala go ominąć.



Ewentualnie można przejść przez cykl.

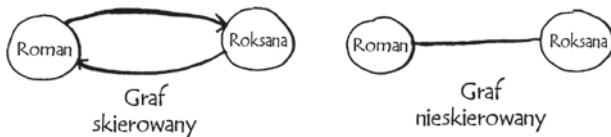


W każdym przypadku kończymy w węźle A, tylko cykl dodaje wartości do wagi sumarycznej. Jeśli chcemy, możemy nawet przejść przez cykl dwa razy.

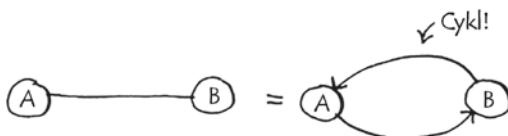


Jednak każda taka wycieczka przez cykl powoduje dodanie 8 do wagi sumarycznej. W związku z tym, przechodząc przez cykl, nigdy nie znajdziemy najkrótszej drogi.

I jeszcze na koniec, może pamiętaś różnicę między grafami skierowanymi i nieskierowanymi omówioną w rozdziale 6.?



Graf nieskierowany to taki, w którym oba węzły wskazują się wzajemnie. A to przecież jest cykl!



W grafie nieskierowanym każda krawędź dodaje kolejny cykl. Algorytm Dijkstry działa tylko z **acyklicznymi grafami skierowanymi** (ang. *directed acyclic graph* — DAG).

Szukanie funduszy na fortepian

Wystarczy tej terminologii, czas obejrzeć kolejny przykład! To jest Roman.

Roman chce przehandlować książkę z nutami za fortepian.

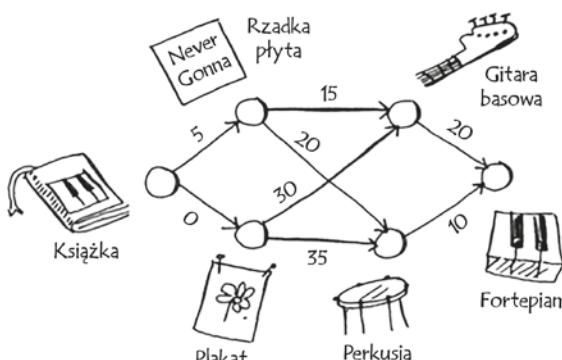


Aleks mówi: „Dam Ci ten plakat za książkę. To plakat mojej ulubionej grupy o nazwie Niszczyciele. Albo mogę dać Ci ten rzadki krążek Ricka Astleya i jeszcze 5 złotych”. „Och, podobno na tej płycie znajduje się świetna piosenka” — mówi Ania. „Dam Ci moją gitarę albo perkusję za plakat albo płytę”.



„Myślałem o kupnie gitary!” — wykrzykuje Bernard. „Hej, dam Ci mój fortepian za którykolwiek z rzeczy Ani”.

Doskonale! Dysponując niewielkimi środkami, Roman zdołał znaleźć sposób, jak zaczynając od książki z nutami utworów na fortepian, dochrapać się prawdziwego fortepianu. Teraz musi tylko dowiedzieć się, co zrobić, aby wydać jak najmniej, by przeprowadzić transakcje. Przedstawimy otrzymane przez niego oferty w postaci grafu.



Wszystkie węzły tego grafu reprezentują przedmioty, którymi może handlować Roman. Wagi przedstawione na krawędziach oznaczają ilość pieniędzy, jaką musiałby zapłacić za te rzeczy. Roman może więc wymienić plakat na gitarę za 30 zł albo zamienić płytę na gitarę za 15 zł. W jaki sposób Roman może znaleźć ścieżkę od książki do fortepianu, która będzie oznaczała konieczność wydania najmniejszej ilości pieniędzy? Ratunkiem jest algorytm Dijkstry! Przypomnij, że algorytm ten składa się z czterech etapów. W tym przykładzie wykonamy wszystkie z nich, a więc obliczymy także ostateczną ścieżkę.

Węzeł	Koszt
Płyta	5
Plakat	0
Gitara	∞
Perkusja	∞
Fortepian	∞

} Jeszcze nie doszliśmy do tych węzłów od początku

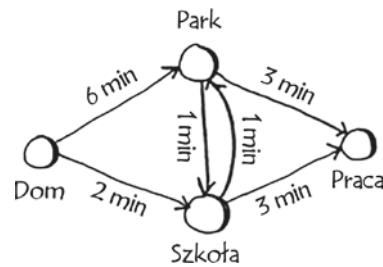
Zanim zaczniemy, musimy zaopatrzyć się w konfigurację początkową. Utworzmy tabelę kosztów wszystkich węzłów. Koszt danego węzła oznacza, ile kosztuje dotarcie do niego.

Tabelę tę będziemy aktualizować podczas działania algorytmu. Aby obliczyć ostateczną ścieżkę, dodatkowo musimy do naszej tabeli dodać kolumnę **rodzic**.

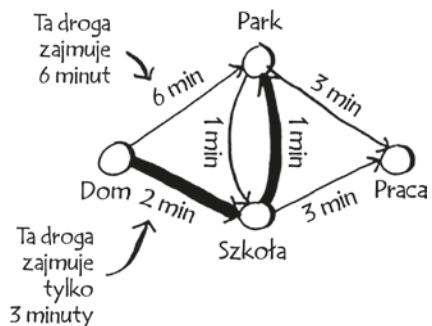
Węzeł	Rodzic
Płyta	Książka
Plakat	Książka
Gitarą	—
Perkusja	—
Fortepeian	—

Niedługo wyjaśnię, o co chodzi z tą kolumną. Na razie jednak uruchomimy algorytm.

Krok 1.: znalezienie najtańszego węzła. W tym przypadku najtańszy jest plakat, ponieważ wymiana na niego kosztuje zero złotych. Czy da się zrobić jeszcze tańszą wymianę? To bardzo ważne, więc dobrze się zastanów. Czy istnieje taka seria transakcji, dzięki której Roman dostanie plakat taniej niż za zero złotych? Gdy już się dobrze zastanowisz, czytaj dalej. Odpowiedź brzmi: nie. *Ponieważ plakat jest najtańszym węzłem dostępnym Romanowi, nie ma sposobu zrobić tego jeszcze taniej.* Można na to spojrzeć także z innej strony. Powiedzmy, że chcemy dojechać z domu do pracy.



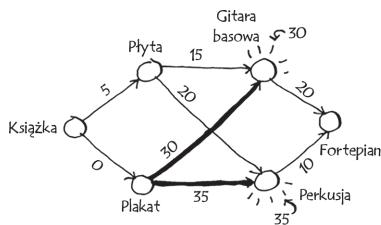
Jeśli pojedziemy w stronę szkoły, podróz zajmie nam dwie minuty. A jeśli pojedziemy w kierunku parku, podróz zajmie nam aż sześć minut. Czy jest taka możliwość, aby pojechać w stronę parku i skończyć przy szkole w czasie krótszym niż dwie minuty? To niemożliwe, ponieważ samo dotarcie do parku trwa już dłużej niż dwie minuty. A czy zatem jest szybsza droga do parku. Tak.



Tak więc wygląda podstawowa idea algorytmu Dijkstry: znajdź najtańszy węzeł w grafie. Nie ma tańszego sposobu dotarcia do tego węzła!

Wracając do naszego muzycznego przykładu, najtańszy do wymiany jest plakat.

Krok 2.: sprawdzenie, ile czasu zajmie dotarcie do sąsiadów (koszt).



Rodzic	Węzeł	Koszt
Książka	Płyta	5
Książka	Plakat	0
Plakat	Gitara	30
Plakat	Perkusja	35
—	Fortepian	∞

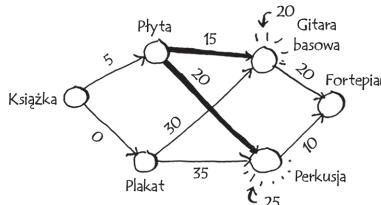
W tabeli przedstawione są ceny gitary basowej i perkusji. Ich wartości zostały ustalone, gdy przeszliśmy do plakatu, więc teraz to właśnie plakat jest ustanowiony jako węzeł nadrzędny (rodzic). Oznacza to, że aby dotrzeć do gitary basowej, musimy pójść krawędzią z plakatu. To samo dotyczy perkusji.

Przechodzimy od „plakatu”, aby dojść do tych węzłów

Rodzic	Węzeł	Koszt
Książka	Płyta	5
Książka	Plakat	0
Plakat	Gitara	30
Plakat	Perkusja	35
—	Fortepian	∞

Krok 1. ponownie: następnym najtańszym węzłem jest płytka, której koszt wynosi 5 zł.

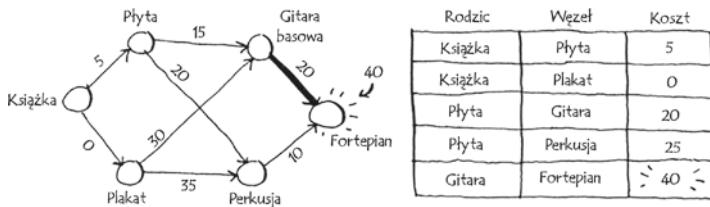
Krok 2. ponownie: aktualizacja wartości wszystkich sąsiadów.



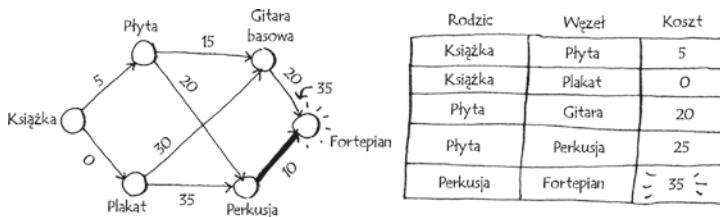
Rodzic	Węzeł	Koszt
Książka	Płyta	5
Książka	Plakat	0
Plakat	Gitara	30
Plakat	Perkusja	25
—	Fortepian	∞

Hej, właśnie zaktualizowaliśmy ceny perkusji i gitary! Znaczy to, że taniej można dojść do perkusji i gitary, idąc krawędzią od płytka. Zatem ustawiamy płytka jako nowego rodzica dla obu instrumentów.

Następnym najtańszym przedmiotem jest gitara basowa. Aktualizujemy jej sąsiadów.

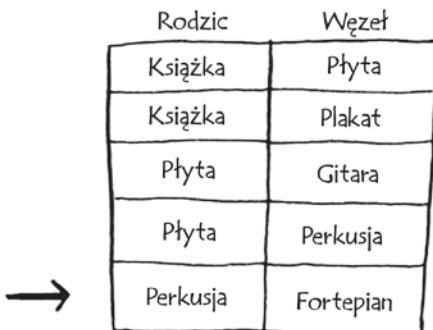


W końcu mamy cenę fortepianu, który możemy wymienić na gitarę. Ustawiamy zatem gitarę jako rodzica. I na koniec ostatni węzeł, czyli perkusja.



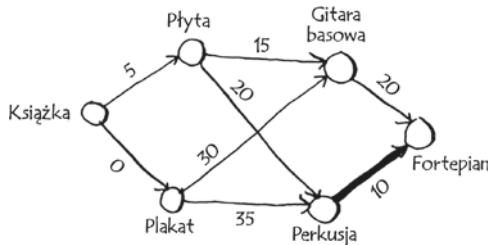
Roman może kupić fortepian jeszcze taniej, jeśli wymieni go na perkusję. Zatem najtańsza możliwa seria transakcji Romana kosztuje 35 zł.

A teraz, zgodnie z obietnicą, określmy drogę. Na razie wiemy, że najkrótsza droga kosztuje 35 zł, tylko jak wyznaczyć ścieżkę? Na początek spójrz na element nadzędny **fortepianu**.

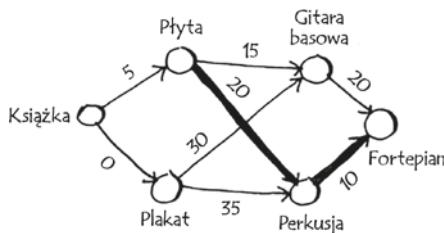


Rodzicem fortepianu jest perkusja. Znaczy to, że Roman wymienia perkusję na fortepian. Podążamy więc tą krawędzią.

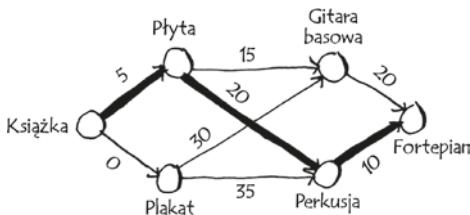
Zobaczmy, jakby to wyglądało na grafie. **Fortepeian** ma rodzica **perkusję**.



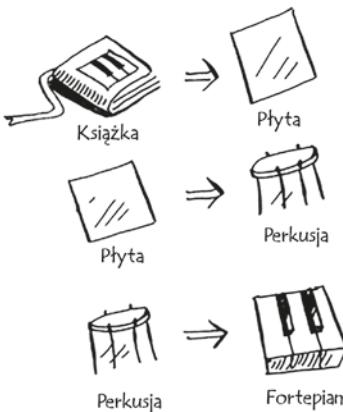
A **rodzicem** perkusji jest płyta.



Zatem Roman wymieni płytę na perkusję. I oczywiście wymieni książkę na płytę. Przechodząc krawędziami wstecz, od rodziców, można odtworzyć kompletną ścieżkę.



Oto seria wymian handlowych, jakie musi wykonać Roman.

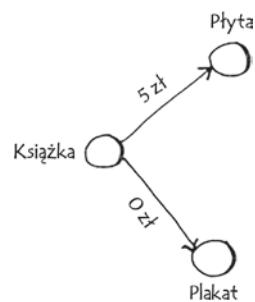
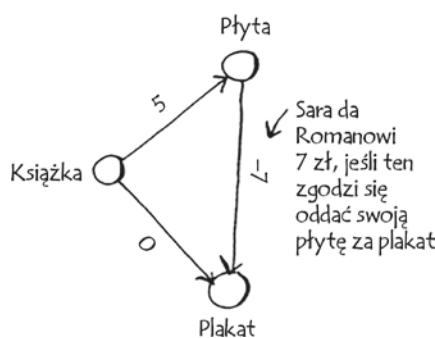


Do tej pory pojęcia *najkrótsza droga* używałem w sensie dosłownym, czyli w znaczeniu najkrótszej ścieżki dzielącej dwa miejsca lub dwie osoby. Mam nadzieję, że ten przykład pozwoli Ci zrozumieć, że najkrótsza droga wcale nie musi oznaczać dystansu fizycznego. Równie dobrze może dotyczyć minimalizacji czegoś innego. W tym przypadku Roman chciał ponieść jak najmniejsze koszty finansowe. Dzięki, Dijkstra!

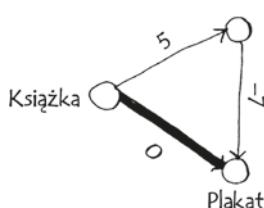
Krawędzie o wagach ujemnych

W poprzednim przykładzie Aleks zaproponował Romanowi dwie rzeczy za jego książkę z nutami.

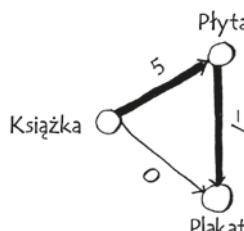
Powiedzmy, że Sara proponuje Romanowi za płytę plakat *i dodatkowo* 7 zł. Taka wymiana Romana nic nie kosztuje, a nawet umożliwia mu zarobienie 7 zł. Jak to przedstawić na grafie?



Krawędź prowadząca od płyty do plakatu ma ujemną wagę! Jeśli Roman zgodzi się na tę propozycję, dostanie 7 zł. Teraz więc Roman może dotrzeć do plakatu na dwa sposoby.

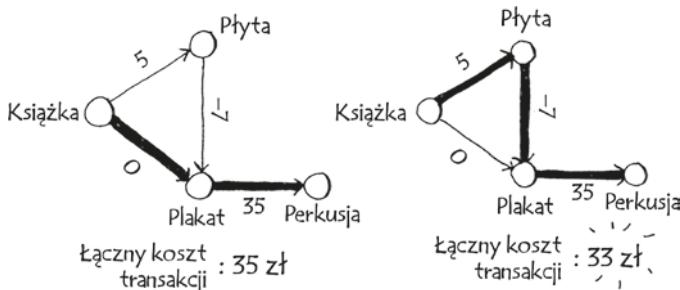


Wybierając tę drogę, Roman dostanie 0 zł



Wybierając tę drogę, Roman odzyska 2 zł

Zatem zgodzenie się na drugą propozycję ma sens — w ten sposób Roman odzyska 2 zł! Przypominam teraz, że Roman może wymienić plakat na perkusję. Ma w związku z tym dwie możliwości do wyboru.

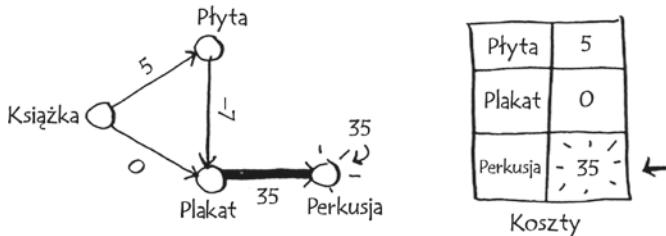


Druga ścieżka kosztuje 2 zł mniej, więc powinien ją wybrać, prawda? Cóż, zgadnij! Jeśli na tym grafie uruchomisz algorytm Dijkstry, Roman wybierze niewłaściwą drogę — tę dłuższą. *Algorytm Dijkstry nie używa się z grafami zawierającymi krawędzie o ujemnej wadze*. Krawędzie tego typu powodują nieprawidłowe działanie tego algorytmu. Zobaczmy, co się stanie, gdy uruchomimy algorytm Dijkstry. Najpierw tworzymy tabelę kosztów.

Płyta	5
Plakat	0
Perkusja	∞

Koszt

Następnie szukamy najtańszego węzła i aktualizujemy koszty jego sąsiadów. W tym przypadku najtańszy jest węzeł plakatu. Zatem wg algorytmu Dijkstry *nie ma tańszego sposobu dotarcia do plakatu niż zapłacenie 0 zł* (wiesz, że to nieprawda!). Mimo to zaktualizujemy koszty sąsiadów.

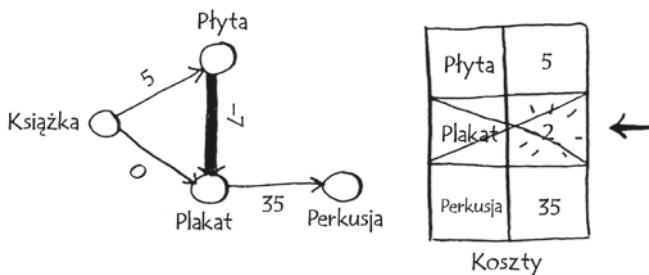


Perkusja kosztuje teraz 35 zł.

Teraz przechodzimy do następnego najbliższego węzła, który jeszcze nie został przetworzony.

Płyta	5
Plakat	0
Perkusja	35

Aktualizujemy koszty sąsiadów.



Już przetworzyliśmy węzeł plakatu, ale i tak aktualizujemy jego koszt. Tu zapala się nam wielka czerwona lampa. Gdy przetworzymy jakiś węzeł, znaczy to, że nie ma już tańszego sposobu na dotarcie do niego. A my właśnie znaleźliśmy tańszy sposób dotarcia do plakatu! Perkusja nie ma sąsiadów, więc to jest koniec algorytmu. Oto ostateczne koszty.

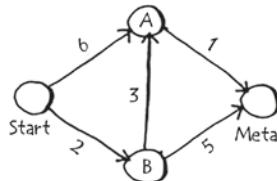
Płyta	5
Plakat	-2
Perkusja	35

Ostateczne koszty

Dojście do perkusji kosztuje 35 zł. Wiemy, że jest droga kosztująca tylko 33 zł, ale algorytm Dijkstry jej nie znalazł. Algorytm założył, że skoro przetwarzaliśmy węzeł plakatu, to nie było szybszego sposobu na dotarcie do niego. Założenie to sprawdza się tylko w przypadkach, gdy nie ma krawędzi o ujemnej wadze. Zatem *algorytm Dijkstry nie działa z krawędziami o ujemnej wadze*. Jeśli trzeba znaleźć najkrótszą drogę w grafie zawierającym krawędzie o ujemnej wadze, należy skorzystać z innego algorytmu! Jest to tzw. **algorytm Bellmana-Forda**. Jego opis wykracza poza zakres tej książeczki, ale łatwo można znaleźć szersze informacje w internecie.

Implementacja

Zobaczmy, jak zaimplementować algorytm Dijkstry za pomocą kodu źródłowego. Poniżej znajduje się graf, którym posłużę się w ramach przykładu.



Do implementacji tego przykładu potrzebne będą trzy tablice skrótów.

	A	B	Meta
Start	6	2	—
A	—	1	—
B	3	—	5
Meta	—	—	—

	A	B	Meta
Start	6	2	∞
A	—	—	—
B	—	—	—
Meta	—	—	—

	A	B	Meta
Start	Start	Start	—
A	—	—	—
B	—	—	—
Meta	—	—	—

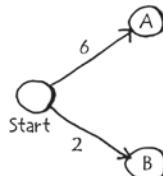
Tablice kosztów i rodziców będziemy aktualizować w miarę postępu wykonywania algorytmu. Najpierw musimy zaimplementować graf. Podobnie jak w rozdziale 6., wykorzystamy do tego celu tablicę skrótów.

```
graph = {}
```

W poprzednim rozdziale wszystkich sąsiadów węzła przechowywaliśmy w tablicy skrótów, tak jak poniżej.

```
graph["ty"] = ["alicja", "bartek", "cecylia"]
```

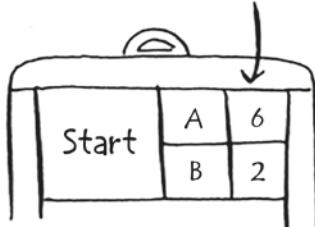
Tym razem jednak musimy zapisać sąsiadów i koszty dotarcia do nich. Węzeł *Start* ma np. dwóch sąsiadów — *A* i *B*.



Jak przedstawić wagę tych krawędzi? Może użyć kolejnej tablicy skrótów?

```
graph["start"] = {}
graph["start"]["a"] = 6
graph["start"]["b"] = 2
```

Ta tabela skrótów
zawiera wewnątrz
inne tablice skrótów



Zatem `graph["start"]` jest tablicą skrótów. Wszystkich sąsiadów węzła startowego można wyświetlić tak.

```
>>> print graph["start"].keys()
["a", "b"]
```

Istnieje po jednej krawędzi prowadzącej od węzła start do węzłów a i b.
A gdybyśmy chcieli sprawdzić wagę wszystkich tych krawędzi?

```
>>> print graph["start"]["a"]
2
>>> print graph["start"]["b"]
6
```

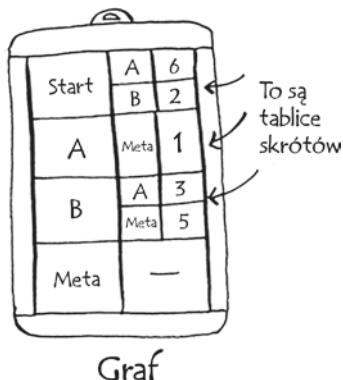
Dodamy pozostałe węzły i ich sąsiadów do grafu.

```
graph["a"] = {}
graph["a"]["meta"] = 1
```

```
graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["meta"] = 5
```

```
graph["meta"] = {} <..... Węzeł mety nie ma żadnych sąsiadów.
```

Teraz cała tablica skrótów reprezentująca graf wygląda tak, jak na poniższym rysunku.



Teraz potrzebujemy tablicy skrótów do zapisania kosztów wszystkich węzłów.

A	6
B	2
Meta	∞

Koszty

Koszt danego węzła oznacza, jaka odległość dzieli go od węzła początkowego. Wiemy, że dotarcie od startu do węzła B zajmuje dwie minuty. Wiemy też, że dotarcie do węzła A z tego samego miejsca zajmuje sześć minut (choć można poszukać krótszej drogi). Nie wiemy natomiast, ile czasu zajmie dotarcie do mety. Gdy koszt jest nieznany, wpisujemy znak nieskończoności. Czy da się przedstawić *nieskończoność* w Pythonie? Oczywiście.

```
infinity = float("inf")
```

Poniżej znajduje się kod tworzący tabelę kosztów.

```
infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
```

Potrzebujemy też jeszcze jednej tablicy do przechowywania rodziców.

A	Start
B	Start
Meta	—

Rodzice

Oto kod tworzący tablicę skrótów do przechowywania rodziców.

```
parents = []
parents["a"] = "start"
parents["b"] = "start"
parents["meta"] = None
```

Na koniec potrzebujemy jeszcze tablicy do rejestrowania sprawdzonych węzłów, aby przez przypadek żadnego z nich nie przetworzyć więcej niż raz.

```
processed = []
```



To cała konfiguracja. Teraz spójrz na schemat działania algorytmu.

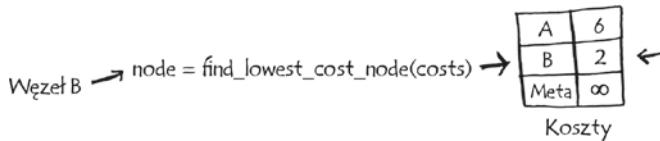
Najpierw przedstawię kod źródłowy, a następnie szczegółowo go opiszę. Oto ten kod.

```

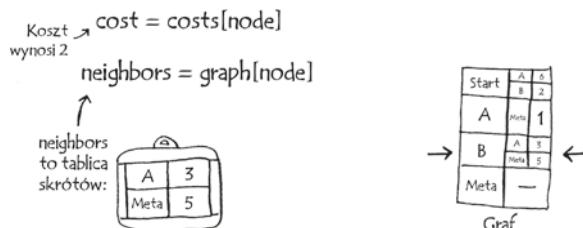
node = find_lowest_cost_node(costs) ← Znajduje najtańszy węzeł, który nie został jeszcze przetworzony.
while node is not None: ← Jeżeli wszystkie węzły zostały przetworzone,
    cost = costs[node] ← następuje zakończenie pętli.
    neighbors = graph[node]
    for n in neighbors.keys(): ← Przegląda wszystkich sąsiadów danego węzła.
        new_cost = cost + neighbors[n] ← Jeżeli dotarcie do tego sąsiada jest tańsze
        if costs[n] > new_cost: ← drogą przez ten węzeł...
            costs[n] = new_cost ← ... zaktualizuj koszt tego węzła.
            parents[n] = node ← Węzeł ten staje się nowym rodzicem tego sąsiada.
    processed.append(node) ← Oznaczenie węzła jako przetworzonego.
    node = find_lowest_cost_node(costs) ← Znajduje następny węzeł do przetworzenia
                                              i wraca na początek pętli.
```

To jest implementacja algorytmu Dijkstry w Pythonie! Kod funkcji pokazuję nieco dalej. A na razie zobaczymy, jak nasz algorytm `find_lowest_cost_node` sprawdzi się w działaniu.

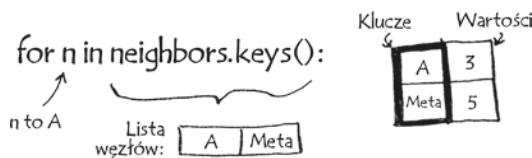
Znalezienie najbliższego węzła.



Pobranie kosztu i sąsiadów tego węzła.



Przeglądanie sąsiadów.



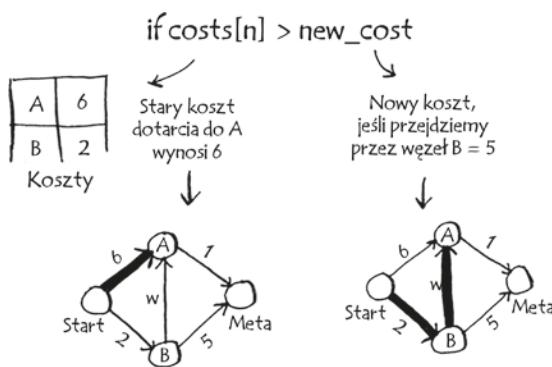
Każdy węzeł ma określony koszt, który wskazuje, ile czasu zajmuje dotarcie do tego węzła od linii startowej. Poniżej obliczamy, ile czasu zajmie dotarcie do węzła A drogą Start -> węzeł B -> węzeł A, zamiast drogą Start -> węzeł A.

$$\text{new_cost} = \text{cost} + \text{neighbors}[n]$$

↑ ↓
Koszt B,
tzn. 2 Odległość
 od B do A = 3

$\left. \begin{array}{c} \text{Klucze} \\ \text{Wartości} \end{array} \right\} \text{new_cost} = 2 + 3 = 5$

Porównajmy te koszty.



Znaleźliśmy krótszą drogę do węzła A! Aktualizujemy koszt.

A	5
B	2
Meta	∞

Koszty

$\text{costs}[n] = \text{new_cost}$

A 5

Nowa droga prowadzi przez węzeł B, więc ustawiamy ten węzeł jako nowego rodzica.

A	B
B	Start
Meta	—

Rodzice

$\text{parents}[n] = \text{node}$

A B

Jesteśmy z powrotem na początku pętli. Następny sąsiad w pętli for to węzeł mety.

for n in neighbors.keys():

n to meta

A	Meta
---	------

Ille czasu zajmie podróż do mety, jeśli pójdziemy przez węzeł B?

$$\text{new_cost} = \text{cost} + \text{neighbors}[n]$$

\downarrow \downarrow

2 Odległość od B
do mety = 5

$$\left. \begin{array}{l} \\ \end{array} \right\} = 7$$

Czas trwania wynosi siedem minut. Poprzednio była nieskończona liczba minut, a siedem minut to z pewnością mniej.

		$\text{if } \text{costs}[n] > \text{new_cost}$
Meta	∞	\downarrow
Koszty		7

Przedtem nie mieliśmy żadnego kosztu do mety

Ustawiamy nowy koszt i nowego rodzica dla węzła mety.

$\text{costs}[n] = \text{new_cost}$	$\begin{array}{ c c } \hline A & 5 \\ \hline B & 2 \\ \hline \text{Meta} & \cancel{7} \\ \hline \end{array}$	\leftarrow
$\begin{array}{cc} \uparrow & \uparrow \\ \text{Meta} & 7 \end{array}$		

Koszty

$\text{parents}[n] = \text{node}$	$\begin{array}{ c c } \hline A & B \\ \hline B & \text{Start} \\ \hline \text{Meta} & \cancel{B} \\ \hline \end{array}$	\leftarrow
$\begin{array}{cc} \uparrow & \uparrow \\ \text{Meta} & B \end{array}$		

Rodzice

Zaktualizowaliśmy koszty wszystkich sąsiadów węzła B, więc możemy go oznaczyć jako przetworzony.

$\text{processed.append(node)}$	$\begin{array}{cc} \text{Przetworzone} \\ \text{węzły: } \boxed{B} \end{array}$
$\begin{array}{c} \uparrow \\ B \end{array}$	

Szukamy następnego węzła do przetworzenia.

$\text{node} = \text{find_lowest_cost_node(costs)}$	$\begin{array}{c} \text{Najtańszy} \\ \text{nieprzetworzony węzeł} \\ \nearrow \\ \boxed{A} \end{array}$	$\begin{array}{ c c } \hline A & 5 \\ \hline \cancel{B} & \cancel{2} \\ \hline \text{Meta} & 7 \\ \hline \end{array}$	\leftarrow
	$\begin{array}{c} \nearrow \\ \text{Już przetworzony} \end{array}$		

Koszty

Pobieramy koszt i sąsiadów węzła A.

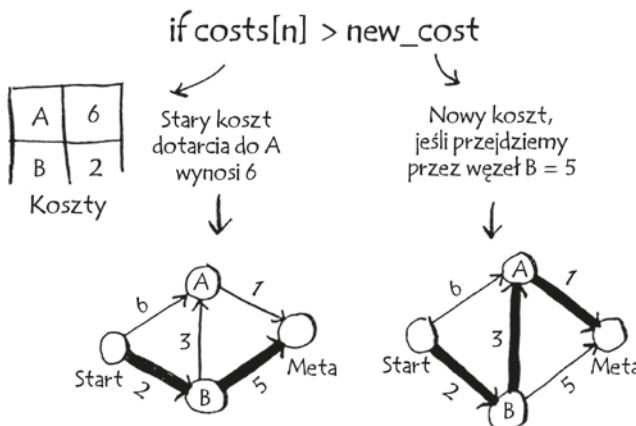
$$\begin{aligned} \text{cost} &= \text{costs}[n] \\ &\uparrow \\ &5 \\ \text{neighbors} &= \text{graph}[node] \\ &\uparrow \\ &\boxed{\text{Meta } 1} \end{aligned}$$

Węzeł A ma tylko jednego sąsiada, którym jest węzeł mety.

```
for n in neighbors.keys():
    ↑
    Meta
    { } ↓
    Meta
```

Aktualnie droga do węzła mety zajmuje siedem minut. Ile czasu byłoby nam potrzebne, gdybyśmy poszli przez węzeł A?

$$\left. \begin{array}{l} \text{new_cost} = \text{cost} + \text{neighbors}[n] \\ \downarrow \qquad \downarrow \\ \text{Koszt dotarcia} \qquad \text{Odległość od A} \\ \text{do A od startu} = 5 \qquad \text{do mety} = 1 \end{array} \right\} \begin{array}{l} 5 + 1 \\ = 6 \end{array}$$



Przez węzeł A szybciej dotrzymy do mety! Aktualizujemy więc koszt i rodzica.

$$\text{costs}[n] = \text{new_cost}$$

	5
A	5
B	2
Meta	6

Koszty

$$\text{parents}[n] = \text{node}$$

A	B
B	Start
Meta	A

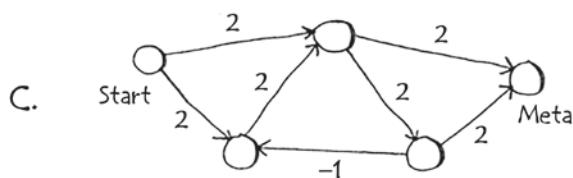
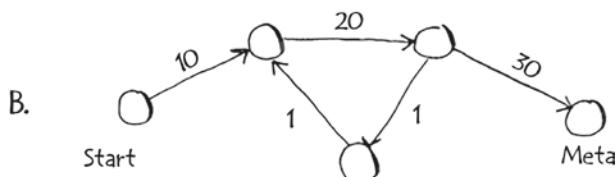
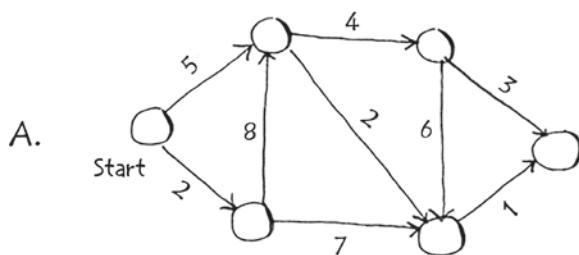
Rodzice

Po przetworzeniu wszystkich węzłów algorytm kończy działanie. Mam nadzieję, że ten szczegółowy opis pomógł lepiej zrozumieć działanie omawianego algorytmu. Z pomocą funkcji `find_lowest_cost_node` szukanie najbliższego węzła jest bardzo łatwe. Oto jej kod źródłowy.

```
def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None
    for node in costs: Przegląda każdy węzeł po kolej.
        cost = costs[node]
        if cost < lowest_cost and node not in
    processed: Jeśli jest to najniższy z dotychczasowych kosztów i nie został jeszcze przetworzony...
            lowest_cost = cost ...ustaw go jako nowy najbliższy węzeł.
            lowest_cost_node = node
    return lowest_cost_node
```

ĆWICZENIE

- 7.1.** Określ wagę najkrótszej drogi od linii startu do mety na każdym z poniższym grafów.



Powtórzenie

- Algorytm wyszukiwania wszerz służy do obliczania najkrótszej drogi w grafie nieważonym.
- Algorytm Dijkstry służy do obliczania najkrótszej drogi w grafie ważonym.
- Algorytm Dijkstry poprawnie działa tylko na grafach zawierających wyłącznie dodatnie wagи.
- Jeśli graf zawiera wagи ujemne, należy posługiwać się algorytmem Bellmana-Forda.



W tym rozdziale:

- dowiesz się, jak dokonać niemożliwego, czyli nauczysz się rozwiązywać zadania, dla których nie istnieją szybkie rozwiązania algorytmiczne (tzw. problemy NP-zupełne),
- nauczysz się rozpoznawać takie problemy, dzięki czemu nie będziesz marnować czasu na szukanie dla nich szybkiego rozwiązania w postaci algorytmu,
- poznasz algorytmy aproksymacyjne, za pomocą których szybko można wyszukiwać przybliżone rozwiązania problemów NP-zupełnych,
- poznasz bardzo prostą strategię zachłanego rozwiązywania problemów.

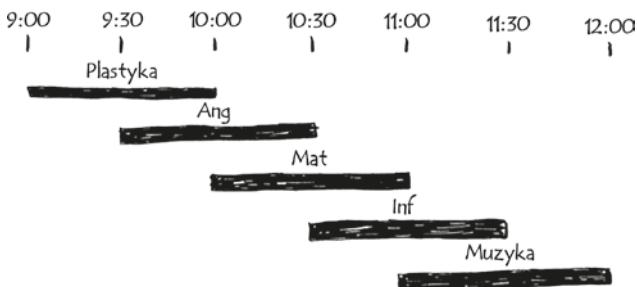
Plan zajęć w sali lekcyjnej

Wyobraź sobie, że dysponujesz salą lekcyjną i chcesz w niej przeprowadzić jak najwięcej lekcji. Oto lista zajęć.

Lekcja	Początek	Koniec
Plastyka	9:00	10:00
Ang	9:30	10:30
Mat	10:00	11:00
Inf	10:30	11:30
Muzyka	11:00	12:00



Wszystkie te zajęcia nie mogą odbyć się w jednej sali, ponieważ niektóre częściowo się pokrywają.



Chcesz przeprowadzić w tej klasie jak najwięcej zajęć. Jak w związku z tym wybierzesz lekcje, aby można było przeprowadzić ich maksymalną możliwą liczbę?

Na pierwszy rzut oka wydaje się, że to trudne zadanie. Jednak algorytm, który je rozwiązuje, jest tak prosty, że aż trudno będzie Ci uwierzyć. Oto on.

1. Wybierz zajęcia, które kończą się najwcześniej. Są to pierwsze zajęcia w sali.
2. Następnie wybierz zajęcia, które zaczynają się po pierwszych zajęciach. Powinieneś wybrać lekcję, która kończy się najwcześniej z wszystkich pozostałych. To będą drugie zajęcia w tej klasie.

Kontynuuj tę procedurę, aż do otrzymania wyniku! Wypróbujemy ten algorytm. Najwcześniej, bo o 10:00, kończy się plastyka, więc wybieramy tę lekcję.

Plastyka	9:00	10:00	✓
Ang	9:30	10:30	
Mat	10:00	11:00	
Inf	10:30	11:30	
Muzyka	11:00	12:00	

Teraz szukamy zajęć, które zaczynają się po 10:00 i kończą najwcześniej z wszystkich pozostałych.

Plastyka	9:00	10:00	✓
Ang	9:30	10:30	✗
Mat	10:00	11:00	✓
Inf	10:30	11:30	
Muzyka	11:00	12:00	

angielski odpada, ponieważ pokrywa się z plastyką, ale matematyka może być.

Na koniec stwierdzamy, że informatyka pokrywa się z matematyką, ale możemy wziąć muzykę.

Plastyka	9:00	10:00	✓
Ang	9:30	10:30	✗
Mat	10:00	11:00	✓
Inf	10:30	11:30	✗
Muzyka	11:00	12:00	✓

Zatem plan zajęć w naszej klasie obejmuje trzy lekcje.



Wielu ludzi mówi, że ten algorytm wydaje się bardzo prosty. Jest zbyt oczywisty, więc coś musi być z nim nie tak. A na tym właśnie polega piękno algorytmów zachłannych — są łatwe! Algorytm zachłanny jest prosty — w każdym kroku wykonaj optymalny ruch. W tym przypadku za każdym razem wybieramy te zajęcia, które kończą się najwcześniej. Mówiąc bardziej fachowo: *w każdym kroku wybieramy lokalnie optymalne rozwiązanie*, dzięki czemu na końcu otrzymujemy globalnie optymalne rozwiązanie. Możesz wierzyć lub nie, ale z pomocą tego prostego algorytmu znajdziesz najlepsze rozwiązanie tego problemu!

Oczywiście algorytmy zachłanne nie zawsze dadzą się zastosować, a szkoda, bo są bardzo łatwe do napisania! Przeanalizujemy jeszcze jeden przykład.

Problem plecaka

Wyobraź sobie, że jesteś zachłannym złodziejem. Wchodzisz do sklepu z plecakiem i masz do wyboru mnóstwo rzeczy, które można ukraść. Musisz jednak ograniczyć się tylko do tych przedmiotów, które zmieszczać się w plecaku o dopuszczalnym obciążeniu 35 kg.

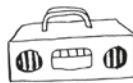


Jeśli chciałbyś nakraść rzeczy za jak największą kwotę, jakiego algorytmu użyłbyś?

Zachłanna strategia i tym razem jest bardzo prosta.

1. Wybierz najdroższą rzecz, którą udźwigniesz w plecaku.
2. Wybierz następną najdroższą rzecz, którą wytrzyma plecak itd.

Jednak tym razem algorytm nas zawiedzie! Wyobraź sobie np., że możesz ukraść trzy produkty.



Sprzęt stereo
Wartość: 3000 zł
30 kg

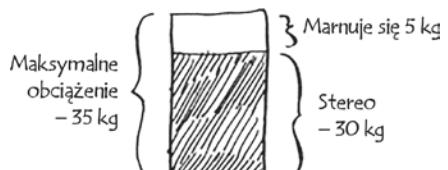


Laptop
Wartość: 2000 zł
20 kg



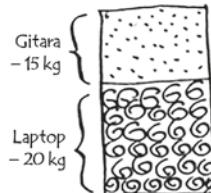
Gitară
Wartość: 1500 zł
15 kg

Twój plecak udźwignie 35 kg towaru. Sprzęt stereo jest najdroższy, więc go bierzesz. Teraz nie ma już miejsca na nic więcej.



Wartość: 3000 zł

Mamy towar za 3000 zł. Zaczekaj! Gdybyśmy zamiast sprzętu stereo wzięli laptop i gitarę, to mielibyśmy towar wart 3500 zł!



Wartość: 3500 zł

Bez wątpienia w tym przypadku zachłanna strategia nie pozwoliła nam uzyskać optymalnego rozwiązania. Jednak byliśmy bardzo blisko. W kolejnym rozdziale wyjaśniam, jak obliczać poprawne rozwiązanie. Jeśli jednak jesteś złodziejem i grasujesz w centrum handlowym, nie przejmujesz się drobiazgami. Wystarczy Ci „dość dobre” rozwiązanie.

Oto morał wynikający z drugiego przykładu: czasami doskonałe jest wrogiem dobrego. W niektórych przypadkach wystarczający jest algorytm rozwiązujący problem „dość dobrze”. I w takich sytuacjach algorytmy zachłanne sprawdzają się doskonale, ponieważ są łatwe w implementacji i zazwyczaj pozwalają uzyskać wynik bardzo zbliżony do optymalnego.

ĆWICZENIA

- 8.1.** Pracujesz w firmie meblarskiej, w której rozwozisz produkty do różnych miejsc w kraju. Musisz załadować swoją ciężarówkę pudłami. Pudła te mają różne rozmiary, więc za każdym razem zastanawiasz się, jak najlepiej wykorzystać dostępną w ciężarówce przestrzeń. Jak ładować pudła, aby spożytkować miejsce do maksimum? Opracuj zachłanną strategię. Czy pozwala ona uzyskać optymalne rozwiązanie?
- 8.2.** Wybierasz się na wycieczkę po Europie i masz siedem dni na obejrzenie jak największej liczby miejsc. Każdemu miejscu przypisujesz wartość punktową (oznaczającą, jak bardzo chcialibyś je odwiedzić) oraz szacujesz, ile czasu zajmie dojazd. W jaki sposób uzyskać jak największą sumę

punktów (aby dotrzeć do wszystkich miejsc, które chce się zobaczyć) podczas wycieczki? Opracuj zachłanną strategię. Czy pozwala ona uzyskać optymalne rozwiązanie?

Przeanalizujemy jeszcze jeden przykład. Będzie to jeden z tych przypadków, w których bez algorytmu zachłanego absolutnie nie można się obejść.

Problem pokrycia zbioru

Załóżmy, że wprowadzamy nową audycję radiową w USA. Naszą ambicją jest dotarcie do wszystkich 50 stanów. Musimy więc wybrać stacje, w których chcielibyśmy być obecni, aby zrealizować założony plan. Czas antenowy w każdej stacji kosztuje, więc staran się też zminimalizować liczbę rozgłośni, za pośrednictwem których będziemy nadawać. Mamy listę stacji.

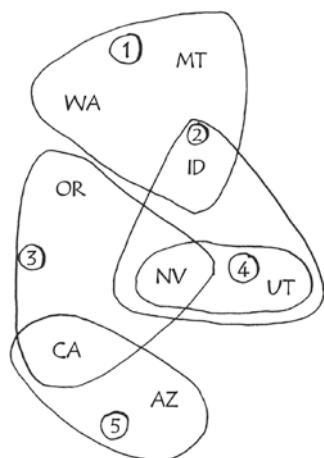
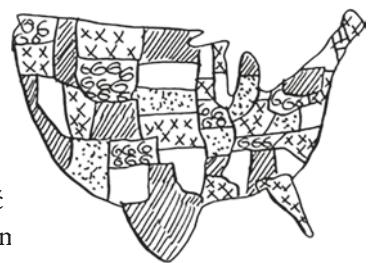
Stacja radiowa	Dostępność
KOne	ID, NV, UT
KTwo	WA, ID, MT
KThree	OR, NV, CA
KFour	NV, UT
KFive	CA, AZ

itd.

Każda stacja pokrywa pewien obszar i obszary różnych stacji wzajemnie się nakładają.

Jak znaleźć najmniejszy możliwy zbiór stacji, w których trzeba wykupić czas antenowy, aby dotrzeć do słuchaczy z wszystkich 50 stanów? Okazuje się, że to bardzo trudne zadanie. A oto procedura.

1. Sporządź listę wszystkich możliwych podzbiorów stacji. Jest to tzw. **zbiór potęgowy**, ponieważ istnieje 2^n możliwych podzbiorów.



Zbiór 1.

...

Zbiór 8.

...

Zbiór 500.



- Z listy wybierz zestaw zawierający najmniejszą liczbę stacji pokrywających wszystkie 50 stanów.

Sęka w tym, że obliczenie wszystkich możliwych podzbiorów stacji zajmuje bardzo dużo czasu. Złożoność obliczeniowa tego zadania wynosi $O(2^n)$, ponieważ liczba podzbiorów wynosi 2^n . Zadanie jest wykonalne przy niewielkiej liczbie stacji, np. 5 czy 10. Jednak wyobraź sobie, co się stanie, kiedy elementów będzie więcej. Jeśli zwiększymy liczbę stacji radiowych, ilość potrzebnego na obliczenia czasu znacząco wzrośnie. Powiedzmy, że możemy obliczyć 10 podzbiorów na sekundę.

Nie ma algorytmu, który wykonywałby te obliczenia wystarczająco szybko! Co w takim razie zrobić?

Liczba stacji	Potrzebny czas
5	3,2 s
10	102,4 s
32	13,6 roku
100	$4 \cdot 10^{21}$ lat

Algorytmy aproksymacyjne

Na ratunek spieszą algorytmy zachłanne! Poniżej opisuję algorytm zachłanny, który znajduje rozwiązanie bardzo bliskie optymalnego.

- Wybierz stację obejmującą największą liczbę stanów, których nie obejmują wybrane wcześniej stacje. Nic się nie stanie, jeśli stacja będzie obejmować niektóre z już uwzględnionych stanów.
- Powtarzaj czynności, aż pokryjesz wszystkie stany.

To się nazywa **algorytm aproksymacyjny**. Jeśli wykonanie dokładnych obliczeń zajmuje za dużo czasu, dobrym rozwiązaniem jest skorzystanie z algorytmu aproksymacyjnego. Algorytmy tego typu ocenia się wg kryteriów:

- szybkości,
- bliskości rozwiązania otrzymywanej do optymalnego rozwiązania.

Algorytmy zachłanne są dobrym wyborem z dwóch powodów. Po pierwsze, łatwo je opracować, a po drugie, ich prostota sprawia, że zazwyczaj dodatkowo są bardzo szybkie. W tym przypadku czas działania algorytmu zachłanego wynosi $O(n^2)$, przy czym n to liczba stacji radiowych.

Zobaczmy, jak opisane rozwiążanie przełożyć na kod źródłowy.

Podstawowy kod źródłowy

W tym przykładzie dla uproszczenia korzystam z podzbioru stanów i rozgłośni radiowych.

Najpierw tworzymy listę stanów, w których chcemy być słyszani.

```
states_needed = set(["mt", "wa", "or", "id", "nv",
"ut",
"ca", "az"]) ←..... Przekazujemy tablicę, która zostaje przekonwertowana na zbiór.
```

W tym przypadku korzystam ze zbioru (`set`). Jest to struktura podobna do listy, w której żaden element nie może się powtarzać. *Innymi słowy, w zbiorze nie może być duplikatów.* Powiedzmy np., że mamy następującą listę.

```
>>> arr = [1, 2, 2, 3, 3, 3]
```

Konwertujemy ją na zbiór.

```
>>> set(arr)
set([1, 2, 3])
```

W zbiorze znajduje się tylko po jednym egzemplarzu liczb 1, 2 i 3.

Potrzebujemy też listy stacji do wyboru. Zapiszemy ją w tablicy skrótów.

```
stations = {}
stations["kone"] = set(["id", "nv", "ut"])
stations["ktwo"] = set(["wa", "id", "mt"])
stations["kthree"] = set(["or", "nv", "ca"])
stations["kfour"] = set(["nv", "ut"])
stations["kfive"] = set(["ca", "az"])
```

Klucze są nazwami stacji, a wartościami są nazwy stanów, które te stacje obejmują. Zatem w przykładzie stacja `kone` obejmuje stany Idaho, Nevada i Utah. Ponadto wszystkie wartości są zbiorami. Wkrótce się przekonasz, że korzystanie z tej struktury znacznie ułatwia życie.

Na koniec potrzebujemy jeszcze czegoś do zapisania wynikowego zbioru stacji.

```
final_stations = set()
```

Obliczanie rozwiązania

Teraz musimy obliczyć, z usług których stacji skorzystać. Spójrz na rysunek po prawej stronie i zastanów się, które stacje — Twoim zdaniem — powinniśmy wybrać.

Poprawnych rozwiązań może być kilka. Musisz przejrzeć wszystkie stacje po kolej i wybrać tę, która obejmuje największą liczbę stanów. Nadamy jej nazwę `best_station`.

```
best_station = None
states_covered = set()
for station, states_for_station in stations.items():
```

Zmienna `states_covered` to zbiór wszystkich obejmowanych przez tę stację stanów, które nie są objęte przez żadną z wcześniejszych stacji. Za pomocą pętli `for` przeglądamy wszystkie stacje w poszukiwaniu tej, która będzie dla nas najlepsza. Przyjrzymy się dokładniej wnętrzu tej pętli.

```
covered = states_needed & states_for_station
if len(covered) > len(states_covered): ←..... Nowa składnia! To się nazywa
    best_station = station
    states_covered = covered
przecięciem zbiorów.
```

W kodzie znajduje się dość niezwykle wyglądający wiersz.

```
covered = states_needed & states_for_station
```

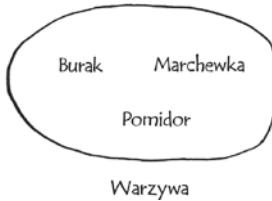
O co w nim chodzi?

Zbiory

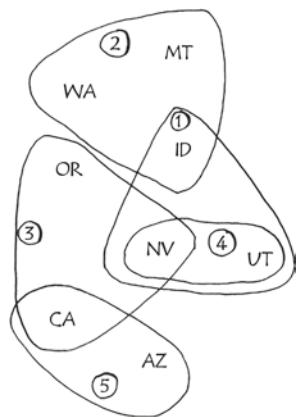
Powiedzmy, że masz zbiór owoców.



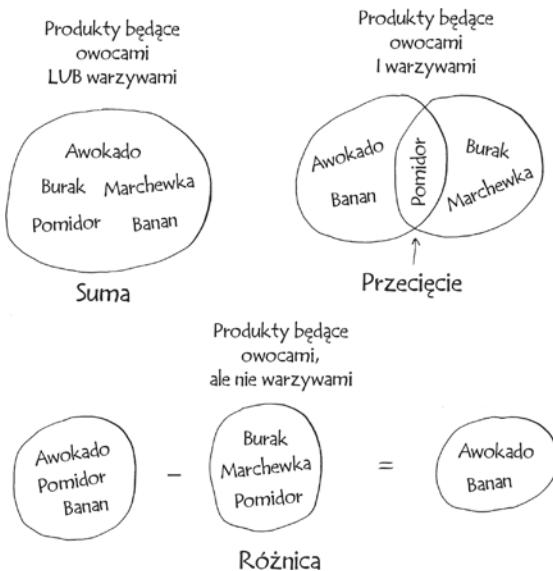
Masz też zbiór warzyw.



Mając dwa zbiory, można z nimi robić różne fajne rzeczy.



Oto niektóre z nich.



- Suma zbiorów to zbiór wszystkich elementów tych zbiorów.
- Przecięcie zbiorów to zbiór elementów, które występują w obu tych zbiorach (w tym przypadku jest to tylko pomidor).
- Różnica zbiorów to zbiór zawierający to, co zostanie po odjęciu elementów jednego zbioru od elementów drugiego zbioru.

Oto przykład.

```
>>> fruits = set(["awokado", "pomidor", "banan"])
>>> vegetables = set(["burak", "marchewka",
"pomidor"])
>>> fruits | vegetables  <..... To jest suma zbiorów.
set(["awocado", "burak", "marchewka", "pomidor",
"banan"])
>>> fruits & vegetables  <..... To jest przecięcie zbiorów.
set(["pomidor"])
>>> fruits - vegetables  <..... To jest różnica zbiorów.
set(["awocado", "banan"])
>>> vegetables - fruits  <..... Jak myślisz, co to będzie?
```

Szybka powtórka.

- Zbiory są jak listy, tylko nie mogą zawierać duplikatów.
- Na zbiorach można wykonywać różne ciekawe działania, np. obliczać sumę, przecięcie i różnicę.

Powrót do kodu

Wracamy do naszego podstawowego przykładu.

To jest przecięcie zbiorów.

```
covered = states_needed & states_for_station
```

Zmienna `covered` jest zbiorem stanów, które znajdowały się zarówno w zbiorze `states_needed`, jak i `states_for_station`. A zatem `covered` jest zbiorem nieobejmowanych stanów, które obejmuje ta stacja! Następnie sprawdzamy, czy stacja ta pokrywa więcej stanów niż aktualnie zapisano w `best_station`.

```
if len(covered) > len(states_covered):
    best_station = station
    states_covered = covered
```

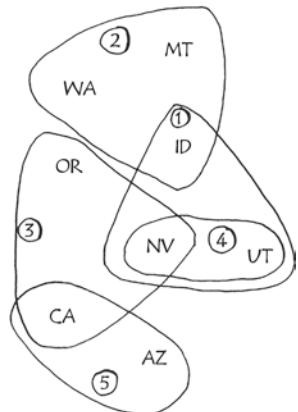
Jeśli tak, to ta stacja jest nową stacją `best_station`. W końcu, jak już pętla `for` zakończy działanie, dodajemy `best_station` do ostatecznej listy stacji.

```
final_stations.add(best_station)
```

Powtarzamy pętlę tyle razy, ile trzeba, aby opróżnić `states_needed`. Oto kompletny kod źródłowy tej pętli.

```
while states_needed:
    best_station = None
    states_covered = set()
    for station, states in stations.items():
        covered = states_needed & states
        if len(covered) > len(states_covered):
            best_station = station
            states_covered = covered

    states_needed -= states_covered
    final_stations.add(best_station)
```



Na koniec możemy wydrukować `final_stations`. Wynik powinien wyglądać tak.

```
>>> print final_stations
set(['ktwo', 'kthree', 'kone', 'kfive'])
```

Czy o to chodziło? Zamiast stacji 1, 2, 3 i 5 moglibyśmy wybrać stacje 2, 3, 4 i 5. Porównajmy czasy wykonywania algorytmów zachłannego i dokładnego.

Liczba stacji	$O(n!)$ Algorytm dokładny	$O(n^2)$ Algorytm zachłanny
5	3,2 s	2,5 s
10	102,4 s	10 s
32	13,6 roku	102,4 s
100	$4 \cdot 10^{21}$ lat	16,67 min

ĆWICZENIA

Dla każdego z tych algorytmów wskaż, czy jest zachłanny, czy nie.

8.3. Szybkie sortowanie.

8.4. Wyszukiwanie wszerz.

8.5. Algorytm Dijkstry.

Problemy NP-zupełne

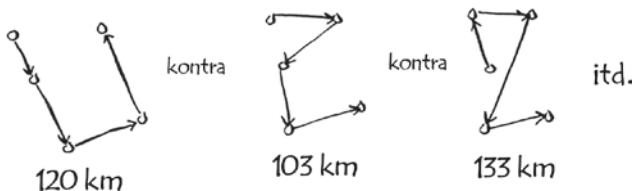
Aby rozwiązać problem pokrycia zbioru, musielibyśmy obliczyć każdy możliwy zbiór.



Może przypomniał Ci się problem komiwojażera z rozdziału 1. W tym przypadku handlowiec ma zawitać do pięciu miast.



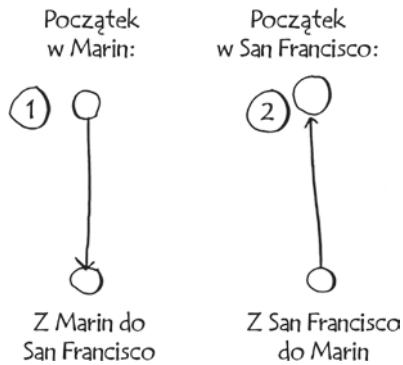
Komiwojażer zastanawia się nad najkrótszą drogą, jaką może obrać, by dotrzeć do każdego z miast. Aby stwierdzić, która trasa jest najkrótsza, najpierw należy obliczyć wszystkie możliwości.



Ile tras trzeba obliczyć w przypadku pięciu miast?

Problem komiwojażera krok po kroku

Zaczniemy od czegoś prostego. Powiedzmy, że mamy odwiedzić tylko dwa miasta, a zatem do wyboru są tylko dwie drogi.



Ta sama czy inna trasa?

Możesz pomyśleć, że w powyższym przykładzie obie możliwości to tak naprawdę jedna trasa. Czy z SF do Marin nie jest tak samo daleko jak z Marin do SF? Niekoniecznie. W niektórych miastach (jak choćby San Francisco) jest wiele dróg jednokierunkowych, przez co nie można wrócić dokładnie tą samą drogą, którą się przyjechało. Poza tym może być konieczne przejedzwanie kilku dodatkowych kilometrów, aby znaleźć wjazd na autostradę, dlatego te dwie trasy nie muszą być identyczne.

Pewnie się zastanawiasz: „Czy w problemie komiwojażera któryś z miast bardziej niż inne zasługuje na to, by od niego zacząć?”. Powiedzmy np., że to ja jestem komiwojażerem. Mieszkam w San Francisco i muszę pojechać do czterech innych miast. W moim przypadku podróż zaczęłaby się w San Francisco.

Czasami jednak miasto początkowe nie jest określone. Wyobraź sobie, że jesteś kurierem FedEx mającym dostarczyć paczkę do miejsca w rejonie zatoki. Przesyłka leci z Chicago do jednego z 50 magazynów FedEx w rejonie zatoki. Następnie pakunek trafi na ciężarówkę, której kierowca dostarcza przesyłki do różnych miejsc. Do którego magazynu powinna trafić paczka? W tym przypadku lokalizacja początkowa jest nieznana. Musisz sam obliczyć optymalną drogę oraz miejsce początku podróży dla komiwojażera.

Czas wykonywania obu wersji jest taki sam. Jednak przykład jest łatwiejszy, jeśli miasto startowe jest niezdefiniowane, więc skorzystam z tej wersji.

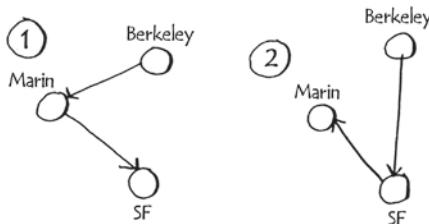
Dwa miasta = dwie możliwe drogi

Trzy miasta

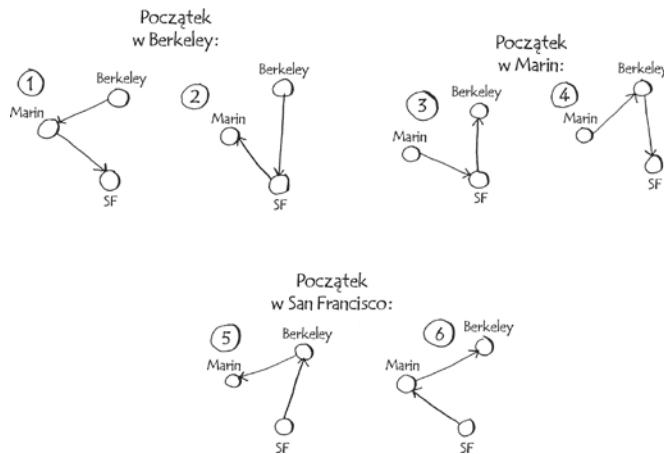
Dodajemy jedno miasto do odwiedzenia. Ile teraz jest możliwych dróg?

Jeśli zaczniemy od Berkeley, do odwiedzenia będą jeszcze dwa miasta.

Początek
w Berkeley:



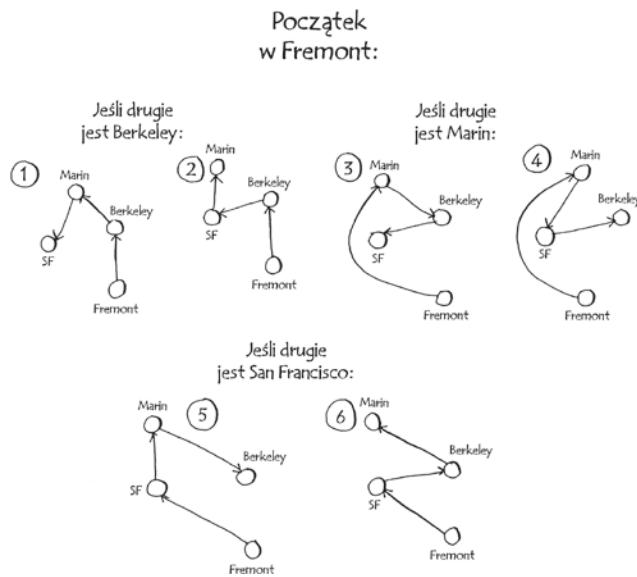
W sumie jest sześć możliwych dróg — po dwie dla każdego miasta, w którym można zacząć podróż.



Zatem trzy miasta oznaczają sześć możliwych tras.

Cztery miasta

Dodajmy kolejne miasto — Fremont — i założymy, że tym razem od niego zaczynamy podróż.



Kiedy zaczynamy od Fremont, mamy sześć możliwych tras. Hej! Te trasy wyglądają tak, jak te obliczone wcześniej, gdy mieliśmy tylko trzy miasta. Tylko w tym przypadku wszystkie trasy zawierają jeszcze dodatkowe miasto — Fremont! Widać tu pewną prawidłowość. Powiedzmy, że mamy cztery miasta i postanawiamy zacząć wędrówkę od Fremont. Pozostają nam więc trzy miasta. A wiemy już, że jeśli są trzy miasta, to liczba wszystkich możliwych dróg między nimi wynosi sześć. Jeśli zaczniemy od Fremont, dalej będziemy mieli sześć możliwości. Oczywiście możemy też zacząć podróżowanie od któregokolwiek z pozostałych miast.

Początek
w Marin:
6 możliwych tras

Początek
w San Francisco:
6 możliwych tras

Początek
w Berkeley:
6 możliwych tras

Cztery miasta, od których można zacząć wędrówkę sześcioma możliwymi trasami, w każdym przypadku dają w sumie $4 * 6 = 24$ możliwe drogi.

Dostrzegasz tę prawidłowość? Za każdym razem, gdy dodajemy miasto, zwiększymy liczbę tras do obliczenia.

Liczba miast	
1	→ 1 droga
2	→ 2 miasta startowe * 1 trasa dla każdego z nich = 2 drogi w sumie
3	→ 3 miasta startowe * 2 trasy = 6 dróg w sumie
4	→ 4 miasta startowe * 6 tras = 24 drogi w sumie
5	→ 5 miast startowych * 24 trasy = 120 dróg w sumie

Ile jest możliwych tras dla sześciu miast? Jeśli powiedziałeś 720, brawo! Dla siedmiu miast jest już 5040, a dla ośmiu aż — 40 320.

Funkcję taką nazywamy **silnią** (może pamiętaś jej opis z rozdziału 3?). Przykładowo $5! = 120$. A teraz wyobraź sobie, że jest 10 miast. Ile jest między nimi możliwych połączeń? $10! = 3\,628\,800$. Dla zaledwie 10 miast trzeba obliczyć aż ponad *trzy miliony* tras. Jak widać, liczba możliwości bardzo szybko

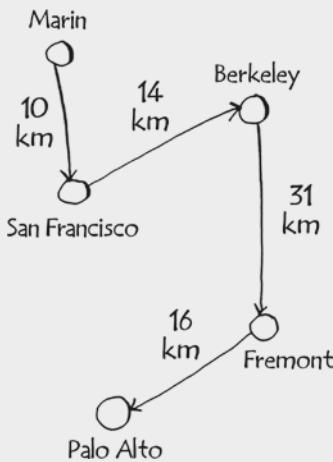
rośnie! Właśnie dlatego niemożliwe jest obliczenie „poprawnego” rozwiązania problemu komiwojażera dla dużej liczby miast.

Problemy komiwojażera i pokrycia zbioru mają jedną cechę wspólną: wymagają obliczenia wszystkich możliwych rozwiązań, aby można było wybrać najmniejszy lub najkrótszy wynik. Oba te problemy są zatem **NP-zupełne**.

Aproksymacja

Jaki jest dobry algorytm aproksymacyjny rozwiązujący problem komiwojażera? Taki, który jest prosty i znajduje krótką drogę. Zastanów się nad odpowiedzią, zanim przeczytasz następne akapity.

Ja zrobiłbym to tak: losowo wybieram miasto startowe. Następnie za każdym razem przy wyborze kolejnego nieodwiedzonego miasta wybieram to, które znajduje się najbliżej. Powiedzmy, że zaczynam podróż od Marin.



Sumaryczna długość trasy wynosi 71 kilometrów. Może nie jest to najkrótsza droga, ale też jest bardzo krótka.

Oto zwięzła definicja NP-zupełności: niektóre problemy słyną z tego, że trudno je rozwiązać. Należą do nich m.in. problemy komiwojażera i pokrycia zbioru. Wiele bardzo inteligentnych osób uważa, że nie da się napisać algorytmu, który pozwalałby szybko rozwiązywać te problemy.

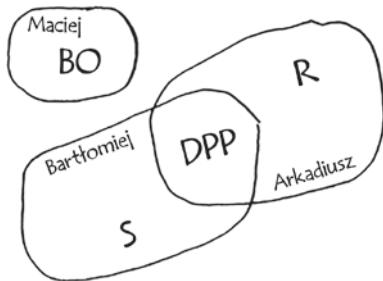
Jak rozpoznać, czy problem jest NP-zupełny

Janek wybiera zawodników do swojej piłkarskiej drużyny marzeń. Przygotował sobie listę pożądanych cech: dobry w rozgrywaniu, szybki, dobrze radzący sobie w deszczu, niepękający pod presją itd. Sporządził też tabelę zawodników, w której każdego zawodnika przyporządkował do posiadanych przez niego umiejętności.



Gracz	Umiejętności
Maciej Flażyński	Boczny obrońca (BO)
Bartłomiej Mieczkowski	Skrzydłowy (S)/dobry pod presją
Arkadiusz Rozbicki	Rozgrywający (R)/dobry pod presją
...	...

Mając na uwadze fakt, że liczebność drużyny jest ograniczona, Janek szuka takiego zespołu, którego członkowie w sumie będą gwarantować wszystkie potrzebne umiejętności. „Chwileczkę — myśli sobie Janek — przecież to jest problem pokrycia zbioru!”.

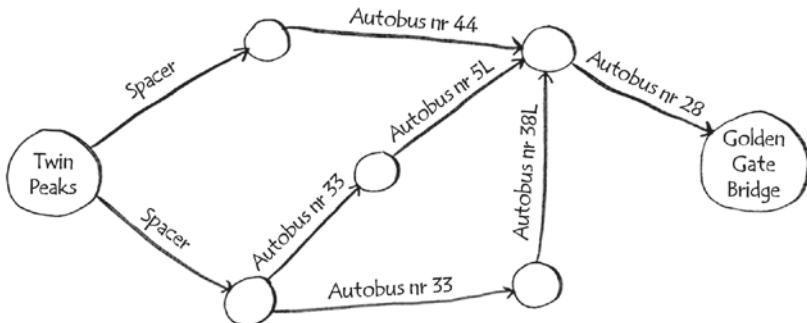


Swoją drużynę Janek może utworzyć za pomocą znanego już algorytmu aproksymacyjnego.

1. Znajdź zawodnika posiadającego najwięcej umiejętności, których nie posiada żaden z wcześniej wybranych zawodników.
2. Powtarzaj tę czynność, aż otrzymasz drużynę spełniającą wszystkie kryteria (albo zabraknie miejsca w zespole).

Problemy NP-zupełne są wszędzie! Warto wiedzieć, czy zadanie, które właśnie próbujemy rozwiązać, nie zalicza się do tej kategorii. Jeśli tak, można zaprzestać

szukania idealnego rozwiązania i zamiast tego wykorzystać algorytm aproksymacyjny. Są w tym, że czasami trudno stwierdzić, czy dany problem jest NP-zupełny. Czasami różnica między prostym problemem a problemem NP-zupełnym jest ledwieauważalna. W poprzednich rozdziałach np. sporo miejsca poświęciłem opisowi metod szukania najkrótszej drogi. Wiesz już, jak obliczyć najkrótszą drogę z punktu A do punktu B.



Gdybyśmy jednak chcieli znaleźć najkrótszą drogę łączącą kilka punktów, wówczas mamy już do czynienia z problemem NP-zupełnym. Mówiąc krótko: nie da się w prosty sposób stwierdzić, czy dany problem jest NP-zupełny. Oto kilka podpowiedzi.

- Twój algorytm działa szybko przy niewielkiej liczbie elementów i radykalnie zwalnia przy większej liczbie elementów.
- „Wszystkie kombinacje X” to często charakterystyczna cecha problemów NP-zupełnych.
- Czy musisz obliczyć „wszystkie możliwe wersje” X, ponieważ nie da się podzielić problemu na mniejsze części? Możliwe więc, że jest to problem NP-zupełny.
- Jeżeli problem dotyczy jakiegoś ciągu (np. listy miast w problemie komiwojażera) i jest trudny do rozwiązania, tzn. że może być NP-zupełny.
- Jeżeli problem dotyczy jakiegoś zbioru (np. zbioru rozgłośni radiowych) i jest trudny do rozwiązania, tzn. że może być NP-zupełny.
- Czy możesz przedstawić zadanie jako problem pokrycia zbioru albo komiwojażera? Jeśli tak, to z pewnością jest to problem NP-zupełny.

ĆWICZENIA

- 8.6.** Listonosz musi dostarczyć listy do 20 domów i chciałby się dowiedzieć, jaka jest najkrótsza trasa między tymi dwudziestoma miejscami. Czy jest to problem NP-zupełny?
- 8.7.** Wyszukiwanie największej kliki w zbiorze ludzi (tu przyjmujemy, że *klika* to zbiór ludzi, którzy wzajemnie się znają) — czy to jest problem NP-zupełny?
- 8.8.** Robisz mapę USA i sąsiadującym stanom chcesz nadać różne kolory. Musisz znaleźć najmniejszą liczbę kolorów, jaka wystarczy do takiego pokolorowania stanów, aby żadna para sąsiadów nie miała tego samego koloru. Czy to jest problem NP-zupełny?

Powtórzenie wiadomości

- Algorytmy zachłanne stosują optymalizację lokalną w nadziei, że uda się obliczyć rozwiązanie optymalne globalnie.
- Nie jest znane żadne szybkie rozwiązanie problemów NP-zupełnych.
- Jeśli trzeba rozwiązać problem NP-zupełny, najlepszym wyjściem jest użycie algorytmu aproksymacyjnego.
- Algorytmy zachłanne są łatwe w implementacji i szybkie, dlatego stanowią dobry wybór.



W tym rozdziale:

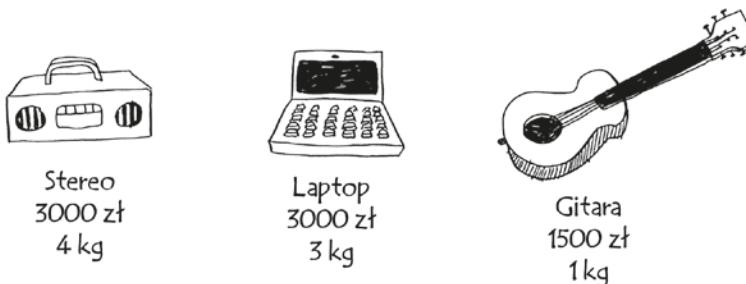
- nauczysz się programowania dynamicznego, czyli techniki rozwiązywania trudnych zadań przez dzielenie ich na mniejsze podzadania i rozwiązywanie najpierw tych mniejszych problemów,
- na podstawie przykładów poznasz metody znajdowania opartych na programowaniu dynamicznym rozwiązań nowego problemu.

Problem plecaka

Wróćmy do przykładu z plecakiem z poprzedniego rozdziału. Jesteś złodziejem i masz plecak, który udźwignie 4 kg produktów.



Do wyboru masz trzy przedmioty, które możesz włożyć do plecaka.



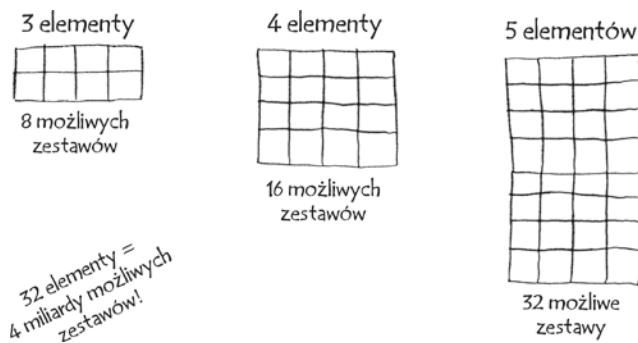
Które przedmioty powinieneś ukraść, aby mieć produkty o jak największej wartości?

Proste rozwiążanie

Najprostsza procedura jest taka: wypróbowujesz każdy możliwy zestaw produktów, aby sprawdzić, który ma największą wartość.



Da się zrobić, ale jest to bardzo czasochłonne. Dla trzech produktów jest 8 możliwych zestawów. Dla czterech produktów jest już 16 zestawów. Każdy kolejny przedmiot powoduje podwojenie liczby zestawów do sprawdzenia! Zatem złożoność obliczeniowa tego algorytmu jest na poziomie $O(2^n)$, a więc jest to bardzo wolny algorytm.



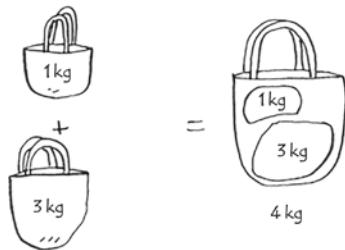
Dla każdej sensownej liczby przedmiotów rozwiązywanie to jest niepraktyczne. W rozdziale 8. pokazałem, jak obliczyć *rozwiązańe przybliżone*, które będzie bliskie optymalnego, ale może nie być idealne.

Jak zatem obliczyć rozwiązanie optymalne?

Programowanie dynamiczne

Odpowiedzią na postawione wyżej pytanie jest programowanie dynamiczne! Zobaczmy, jak w tym przypadku sprawdzi się algorytm programowania dynamicznego. W technice tej najpierw problem dzieli się na podproblemy, a następnie przechodzi do rozwiązywania coraz większych części.

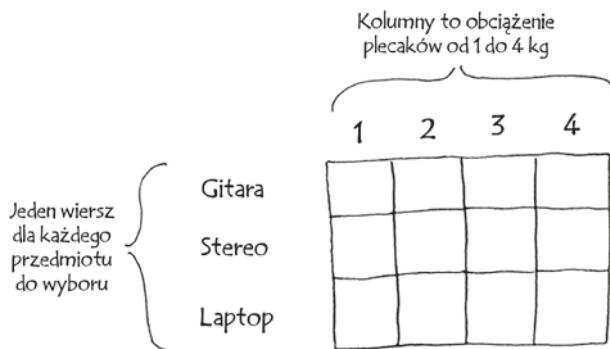
Problem z plecakiem zaczniemy rozwiązywać od mniejszych plecaków („podplecaków”), które następnie będziemy zwiększać, aż rozwiążemy oryginalne zadanie.



Programowanie dynamiczne to trudna koncepcja, więc nie przejmuj się, jeśli nie zrozumiesz wszystkich zasad za pierwszym razem. Dla ułatwienia przytaczam wiele przykładów.

Na początek pokazuję algorytm w akcji. Gdy prześledzisz jego działanie, z pewnością będziesz mieć wiele pytań! Postaram się na nie odpowiedzieć.

Każdy algorytm programowania dynamicznego na początku tworzy siatkę. Poniżej znajduje się siatka dla problemu z plecakiem.



Wiersze tej siatki reprezentują produkty, a kolumny — obciążenie plecaka w przedziale od 1 do 4 kg. Wszystkie te kolumny są potrzebne, ponieważ przy ich użyciu będą obliczane wartości podplecaków.

Początkowo siatka jest pusta, ale zapełnimy każdą jej komórkę. Gdy siatka stanie się pełna, będziemy mieć rozwiązanie naszego problemu! Wykonuj zadania razem ze mną. Utwórz własną siatkę i będziemy ją wypełniać wspólnie.

Wiersz gitary

Dokładny sposób obliczania wartości tej siatki przedstawiam dalej w tym rozdziale. Teraz krok po kroku zapełnimy naszą siatkę, zaczynając od pierwszego wiersza.

	1	2	3	4
Gitara				
Stereo				
Laptop				

Pierwszy jest wiersz *gitary*, co oznacza, że próbujemy włożyć do plecaka ten instrument. W każdej komórce do podjęcia jest prosta decyzja: kradniemy gitarę czy nie? Przypomnę, że chodzi o wybór jak najcenniejszego zestawu produktów.

Pierwsza komórka reprezentuje plecak o obciążeniu 1 kg. Gitara waży właśnie tyle, więc zmieści się do tego plecaka! Zatem komórka ta ma wartość 1500 zł i zawiera gitarę.

Wstawiamy więc do siatki pierwszą wartość.

	1	2	3	4
Gitara	1500 zł G			
Stereo				
Laptop				

W analogiczny sposób wypełnimy wszystkie pozostałe komórki siatki, tworząc listę elementów, które zmieszczą się w plecaku.

Spójrzmy na następną komórkę. Reprezentuje ona plecak o obciążeniu 2 kg. Bez wątpienia gitara się zmieści!

	1	2	3	4
Gitara	1500 zł G	1500 zł G		
Stereo				
Laptop				

To samo robimy w stosunku do pozostałych komórek w pierwszym wierszu. Przypomnę, że to jest pierwszy wiersz, więc do wyboru jest *tylko* gitara. Udajemy, że w tej chwili pozostałych dwóch przedmiotów nie da się przywłaszczyć.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo				
Laptop				

Teraz pewnie się zastanawiasz, *po co* rozważać te wszystkie plecaki o obciążeniach 1 kg, 2 kg itd., skoro zadanie dotyczy plecaka o udźwigu 4 kg? Pamiętasz, jak napisałem, że w programowaniu dynamicznym pracę zaczyna się od małego problemu i stopniowo dochodzi do większego? Teraz rozwiążujemy podproblemy, aby umożliwić sobie rozwiązanie całego problemu. Czytaj dalej, a wkrótce wszystko stanie się jasne.

W tym momencie Twoja siatka powinna wyglądać tak.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo				
Laptop				

Pamiętaj, że Twoim celem jest maksymalizacja wartości towaru w plecaku. *Ten wiersz reprezentuje aktualnie najlepszy wybór, umożliwiający osiągnięcie tego celu.* W związku z tym, wg tego wiersza, gdybyśmy mieli plecak o obciążeniu 4 kg, maksymalnie moglibyśmy włożyć do niego towar wart 1500 zł.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo				
Laptop				

Nasz aktualny najlepszy wybór dla złożenia: gitara za 1500 zł

Oczywiście wiesz, że to nie jest jeszcze ostateczne rozwiązanie. W miarę wykonywania algorytmu będziemy korygować nasze szacunki.

Wiersz stereo

Przechodzimy do następnego wiersza, reprezentującego stereo. W tym wierszu mamy do dyspozycji stereo i gitarę. W każdym wierszu możemy ukraść przedmiot reprezentowany przez ten wiersz i wszystkie poprzednie. Nie możemy zatem jeszcze na razie wziąć laptopa, ale możemy przywlaszczyć sobie stereo i (lub) gitarę. Zajrzyjmy do pierwszej komórki, która reprezentuje plecak o obciążeniu 1 kg. Aktualnie w plecaku o tym obciążeniu jesteśmy w stanie maksymalnie pomieścić towar za 1500 zł.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo				
Laptop				

Aktualne maksimum dla plecaka o obciążeniu 1 kg

Nowe maksimum dla plecaka o obciążeniu 1 kg

Powinniśmy wziąć stereo, czy nie?

W plecaku „udźwigniemy” 1 kg towaru. Czy to wystarczy, aby włożyć do niego stereo? Nie, ten sprzęt jest za ciężki! Skoro nie możemy włożyć stereo, to dla plecaka o obciążeniu 1 kg najcenniejszym wyborem *pozostaje* gitara za 1500 zł.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G			
Laptop				

To samo dotyczy dwóch następnych komórek, które reprezentują plecaki o obciążeniach 2 i 3 kg. Stara maksymalna wartość obu tych komórek wynosiła 1500 zł.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	
Laptop				

Stereo nadal się nie mieści, więc nic się nie zmienia w zakresie najwyższej wartości towaru.

A jak jest w przypadku plecaka o obciążeniu 4 kg? Aha — w końcu można włożyć stereo! Poprzednia maksymalna wartość wynosiła 1500 zł, ale jeśli włożymy do plecaka stereo zamiast gitary, wartość wzrośnie do 3000 zł! Bierzymy więc sprzęt grający.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop				

Właśnie zaktualizowaliśmy szacowaną wartość! Jeżeli obciążenie naszego plecaka wynosi 4 kg, to możemy do niego włożyć towar za przynajmniej 3000 zł. Na siatce widać, że stopniowo zmieniamy nasze szacunki.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop				

← Stara szacunkowa wartość
 ← Nowa szacunkowa wartość
 ← Ostateczne rozwiązańe

Wiersz laptopa

Powtarzamy poprzednie czynności dla laptopa! Ten przenośny komputer waży 3 kg, więc nie możemy go włożyć do plecaka o obciążeniu 1 ani 2 kg. W związku z tym szacunkowa maksymalna wartość dwóch komórek pozostaje na poziomie 1500 zł.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop	1500 zł G	1500 zł G		

W plecaku o obciążeniu 3 kg był towar o wartości 1500 zł, ale teraz możemy do niego włożyć laptop, który kosztuje 2000 zł. Zatem nowa maksymalna wartość w tym plecaku wynosi 2000 zł!

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop	1500 zł G	1500 zł G	2000 zł L	

Najciekawsze rzeczy są jednak dopiero w plecaku o obciążeniu 4 kg. To ważny moment. Aktualnie maksymalna wartość tego plecaka wynosi 2000 zł. Możemy do niego włożyć laptop, ale ten kosztuje tylko 2000 zł.

$$\begin{array}{ccc} 3000 \text{ zł} & \text{kontra} & 2000 \text{ zł} \\ \text{Stereo} & & \text{Laptop} \end{array}$$

Znaczy to, że nowa wartość jest gorsza od poprzedniej. Ale chwileczkę! Laptop waży tylko 3 kg, więc mamy jeszcze 1 kg do wykorzystania! Możemy spróbować coś jeszcze dorzucić.

$$\begin{array}{ccc} 3000 \text{ zł} & \text{kontra} & \left(2000 \text{ zł} + \frac{\text{??}}{\text{Maks. wartość dla } 1\text{ kg}} \right) \\ \text{Stereo} & & \end{array}$$

Jaką maksymalnie wartość możemy zyskać na 1 kg wagi? Już to liczyliśmy.

1	2	3	4
1500 zł G	1500 zł G	1500 zł G	1500 zł G
↓	↓	↓	↓
1500 zł G	1500 zł G	1500 zł G	3000 zł S

Maksymalna wartość dla 1 kg →

Według ostatniego oszacowania najwyższej możliwej wartości, w tej pozostałości przestrzeni powinna zmieścić się gitara, która jest warta 1500 zł. Zatem porównujemy następujące dwie wartości.

$$\begin{array}{ccc} 3000 \text{ zł} & \text{kontra} & \left(2000 \text{ zł} + 1500 \text{ zł} \right) \\ \text{Stereo} & & \text{Laptop} \end{array}$$

Pewnie zastanawiałeś się, po co obliczać maksymalne wartości mniejszych plecaków. Mam nadzieję, że teraz stało się to jasne! Kiedy zostanie trochę miejsca, rozwiązania podproblemów można wykorzystać do jego zapełnienia. Okazuje się, że najlepiej wziąć laptop za 3500 zł.

Ostatecznie nasza siatka wygląda tak, jak na poniższym rysunku.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop	1500 zł G	1500 zł G	2000 zł L	3500 zł L G

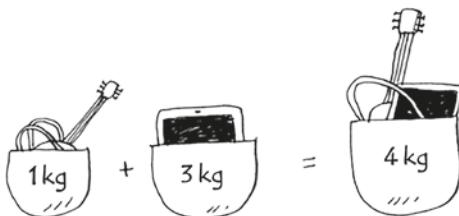
↑
Odpowiedź

Oto odpowiedź: w plecaku zmieści się towar maksymalnie za 3500 zł, czyli gitara i laptop!

Może myślisz sobie, że wartość ostatniej komórki obliczyłem przy użyciu innego wzoru. Można odnieść takie wrażenie, ponieważ pominąłem część nieistotnych spraw związanych z wstawianiem wartości do poprzednich komórek. Wartość każdej komórki jest obliczana za pomocą tego samego wzoru, pokazanego poniżej.

Wiersz Kolumna
 komórka[i][j] = maks. z $\begin{cases} 1. \text{poprzednie maks. (wartość w komórce [i-1][j])} \\ \text{kontra} \\ 2. \text{wartość bieżącego elementu + wartość pozostałego miejsca} \\ \quad \uparrow \\ \text{komórka[i-1][j-waga elementu]} \end{cases}$

Przy użyciu tego wzoru można obliczyć wartość dla każdej komórki siatki i ostatecznie powinno się otrzymać taką samą siatkę jak moja. Pamiętasz, co napisałem o rozwiązywaniu podproblemów? Tutaj połączymy rozwiązania dwóch mniejszych problemów, aby otrzymywać rozwiązanie jednego większego.



Pytania dotyczące problemu plecaka

Jeśli opisane rozwiązywanie wydaje Ci się magiczne, poniżej zamieszczam odpowiedzi na niektóre najczęściej zadawane pytania.

Co się dzieje, gdy zostanie dodany element

Powiedzmy, że nagle odkrywamy, że jest jeszcze jeden produkt, który można ukraść, a którego wcześniej nie zauważylismy! Okazuje się, że możemy sobie przywaszczyć iPhone'a.

Czy musimy wykonać wszystkie obliczenia od początku, aby uwzględnić ten nowy przedmiot? Nie. Przypominam, że w programowaniu dynamicznym stopniowo buduje się tablicę szacunkowych wartości. Na razie mamy obliczone następujące maksymalne wartości.



iPhone
2000 zł
1kg

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop	1500 zł G	1500 zł G	2000 zł L	3500 zł L G

Znaczy to, że do plecaka o obciążeniu 4 kg możemy zapakować maksymalnie towar o wartości 3500 zł. Myśleliśmy, że to jest ostateczna maksymalna wartość. Jednak teraz dodajemy jeszcze jeden wiersz dla iPhone'a.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop	1500 zł G	1500 zł G	2000 zł L	3500 zł L G
iPhone				

↗ Nowa odpowiedź

Wygląda na to, że mamy nową maksymalną wartość! Spróbuj samodzielnie wypełnić nowy wiersz, zanim przeczytasz resztę tekstu.

Zaczniemy od pierwszej komórki. iPhone zmieści się w plecaku o obciążeniu do 1 kg. Poprzednia maksymalna wartość wynosiła 1500 zł, ale iPhone jest wart 2000 zł, więc teraz bierzemy go zamiast gitary.

	1	2	3	4
Gitara	1500 zł G	1500 zł G	1500 zł G	1500 zł G
Stereo	1500 zł G	1500 zł G	1500 zł G	3000 zł S
Laptop	1500 zł G	1500 zł G	2000 zł L	3500 zł L G
iPhone	2000 zł I			

W następnej komórce zmieścimy iPhone i gitarę.

1500 zł G	1500 zł G	1500 zł G	1500 zł G
1500 zł G	1500 zł G	1500 zł G	3000 zł S
1500 zł G	1500 zł G	2000 zł L	3500 zł L G
2000 zł I	3500 zł I G		

W trzeciej komórce nie wymyślimy nic lepszego niż ponowne włożenie iPhone'a i gitary, więc nic nie zmieniamy.

W ostatniej komórce znowu robi się ciekawie. Aktualna maksymalna wartość wynosi 3500 zł. Jednak możemy wziąć iPhone i wówczas do dyspozycji będziemy mieli jeszcze 3 kg.

$$\begin{array}{l} \text{3500 zł kontra} \\ \text{Laptop + gitara} \end{array} \left(\begin{array}{l} \text{2000 zł iPhone} \\ + \end{array} \right) \overline{\begin{array}{l} \text{???} \\ \text{3 kg wolnego miejsca} \end{array}}$$

Te trzy kilogramy są warte 2000 zł! Mamy 2000 zł za iPhone + 2000 zł ze stnego podproblemu, co w sumie daje nam 4000 zł. Nową maksymalną wartość!

Oto nowa kompletna siatka.

1500 zł G	1500 zł G	1500 zł G	1500 zł G
1500 zł G	1500 zł G	1500 zł G	3000 zł S
1500 zł G	1500 zł G	2000 zł L	3500 zł L G
3000 zł I	3500 zł I G	3500 zł I G	4000 zł I L

↑
Nowa odpowiedź

Czy wartość którejkolwiek z kolumn może się obniżyć? Czy jest to możliwe?

Maksymalna wartość spadła podczas dalszej analizy ↓

1	2	3	4
1500 zł	1500 zł	1500 zł	1500 zł
O	O	O	3000 zł

Zastanów się nad tym, zanim przeczytasz moją odpowiedź.

Odpowiedź brzmi: nie. W każdej iteracji zapisujemy aktualną maksymalną szacunkową wartość. Szacunki nie mogą stać się mniej korzystne niż były poprzednio!

ĆWICZENIE

- 9.1.** Powiedzmy, że możemy ukraść jeszcze jeden produkt — odtwarzacz MP3. Urządzenie waży 1 kg i kosztuje 1000 zł. Czy warto je brać?

Jaki będzie skutek zmiany kolejności wierszy

Czy odpowiedź się zmieni? Wyobraź sobie, że wiersze są ustawione w następującej kolejności: stereo, laptop, gitara. Jak będzie wyglądała siatka? Wypełnij ją samodzielnie, zanim przeczytasz moją odpowiedź.

Teraz siatka będzie wyglądać tak, jak na poniższym rysunku.

	1	2	3	4
Stereo	O	O	O	3000 zł S
Laptop	O	O	2000 zł L	3000 zł S
Gitara	1500 zł G	1500 zł G	2000 zł L	3500 zł LG

Odpowiedź się nie zmieniła, a więc kolejność wierszy nie ma znaczenia.

Czy siatkę można wypełniać wg kolumn zamiast wierszy

Spróbuj! W opisywanym przypadku to nie sprawia różnicy, ale w innych przypadkach może mieć znaczenie.

Co się stanie, gdy doda się mniejszy element

Powiedzmy, że możemy ukraść naszyjnik, który waży 0,5 kg i kosztuje 1000 zł. Do tej pory przy tworzeniu siatek zakładaliśmy, że wagę są wyrażone w liczbach całkowitych. Teraz postanawiamy ukraść naszyjnik, więc pozostaje nam jeszcze 3,5 kg. Jaka jest maksymalna wartość towarów, którymi możemy zapełnić pozostałe miejsca? Nie wiadomo! Obliczenia wykonywaliśmy tylko dla plecaków o obciążeniach 1, 2, 3 i 4 kg, a teraz chcemy się dowiedzieć, jaka jest maksymalna wartość plecaka o obciążeniu 3,5 kg.

Ze względu na naszyjnik musimy zwiększyć rozdzielcość siatki.

	0,5	1	1,5	2	2,5	3	3,5	4
Gitara								
Stereo								
Laptop								
Naszyjnik								

Czy można ukraść ułamek przedmiotu

Wyobraź sobie, że jesteś złodziejem grasującym w sklepie spożywczym. Możesz kraść worki soczewicy i ryżu. Jeśli w plecaku nie mieści się cały worek, można go otworzyć i wziąć tylko tyle ziarna, ile się zmieści. Teraz to już nie jest sytuacja typu wszystko albo nic, ponieważ produkty możemy dzielić. Jak sobie z tym poradzić w programowaniu dynamicznym?

Odpowiedź: nie da się. W rozwiążaniu opartym na programowaniu dynamicznym dany przedmiot można wziąć lub nie. Nie ma możliwości stwierdzenia, że w tym przypadku lepiej wziąć tylko połowę produktu.

Jednak przypadek ten można łatwo rozwiązać za pomocą algorytmu zachłanego! Najpierw bierzemy jak najwięcej najcenniejszego produktu. Gdy ten się skończy, bierzemy jak najwięcej drugiego najcenniejszego produktu itd.

Powiedzmy np., że możemy wybierać spośród tych trzech produktów.



Kilogram komosy kosztuje najwięcej z tych trzech produktów, więc bierzemy jej tyle, ile damy rady unieść! Jeśli uda się nią napełnić cały plecak, będzie to najlepsze rozwiązanie.

Jeśli po wsypaniu całej komosy w plecaku zostanie jeszcze trochę miejsca, należy wziąć następny najcenniejszy produkt itd.



Optymalizacja planu podróży

Wyobraź sobie, że jedziesz do Londynu na zaszużone wakacje. Spędzisz tam dwa dni i w tym czasie chcesz zrobić wiele rzeczy. Nie uda Ci się zrealizować wszystkich planów, więc przygotowujesz listę atrakcji.

Atrakcja	Czas	Ocena
Opactwo Westminsterskie	½ dnia	7
Globe Theatre	½ dnia	6
National Gallery	1 dzień	9
Muzeum Brytyjskie	2 dni	9
Katedra św. Pawła	½ dnia	8

Obok każdego miejsca, które chcesz zwiedzić, wpisaliś ilość potrzebnego czasu i ocenę, jak bardzo chciałbyś je zobaczyć. Czy na podstawie tej listy jesteś w stanie stwierdzić, co powinieneś obejrzeć?

To też jest problem tego samego typu, co z plecakiem! Tylko zamiast plecaka mamy ograniczoną ilość czasu. A zamiast sprzętów grających i komputerów mamy listę miejsc do zwiedzenia. Narysuj siatkę programowania dynamicznego dla tego zadania, zanim przeczytasz dalszy tekst.

Siatka powinna wyglądać tak, jak na poniższym rysunku.

	$\frac{1}{2}$	1	$1\frac{1}{2}$	2
Opactwo Westminsterskie				
Globe Theatre				
National Gallery				
Muzeum Brytyjskie				
Katedra św. Pawła				

Czy prawidłowo skonstruowałeś siatkę? Jeśli tak, to ją wypełnij. Które miejsca koniecznie powinieneś zwiedzić? Oto odpowiedź.

	$\frac{1}{2}$	1	$1\frac{1}{2}$	2
Opactwo Westminsterskie	7 O	7 O	7 O	7 O
Globe Theatre	7 O	13 OG	13 OG	13 OG
National Gallery	7 O	13 OG	16 ON	22 OGN
Muzeum Brytyjskie	7 O	13 OG	16 ON	22 OGN
Katedra św. Pawła	8 OK	15 OK	21 OGK	24 ONK

↑
Odpowiedź: Opactwo Westminsterskie, National Gallery, Katedra św. Pawła

Postępowanie z wzajemnie zależnymi przedmiotami

Powiedzmy, że chcemy pojechać do Paryża i w związku z tym dodajemy do listy kilka pozycji.

Te miejsca zajmują dużo czasu, ponieważ aby je zwiedzić, trzeba dojechać z Paryża do Londynu. To zajmuje pół dnia. Gdybyśmy chcieli zająrzeć w każde z tych trzech miejsc, zajęłoby to cztery i pół dnia.

Wieża Eiffla	1½ dnia	8
Luwr	1½ dnia	9
Notre Dame	1½ dnia	7

Chwileczkę, to nieprawda. Nie musimy jechać do Paryża za każdym razem. Gdy już dotrzemy do tego miasta, zwiedzenie każdej atrakcji zajmie tylko po jednym dniu. W związku z tym powinniśmy policzyć po dniu za każdą pozycję na liście plus pół dnia na podróż do Paryża, co w sumie daje 3,5 a nie 4,5 dnia.

Jeśli włożymy Wieżę Eiffla do naszego plecaka, to Luwr stanie się „tańszy” — będzie kosztował dzień zamiast półtora dnia. Jak to przedstawić w postaci modelu w programowaniu dynamicznym?

Nie da się. Siłą programowania dynamicznego jest rozwiązywanie podproblemów i na podstawie ich rozwiązań znajdowanie rozwiązań większych problemów. *Programowanie dynamiczne można stosować tylko w odniesieniu do podproblemów dyskretnych — tzn. takich, które nie są od siebie wzajemnie zależne.* Znaczy to, że w algorytmie programowania dynamicznego nie da się uwzględnić Paryża.

Czy możliwe jest, aby rozwiązanie wymagało więcej niż dwóch podplecaków

Istnieje możliwość, że optymalne rozwiązanie będzie wymagało kradzieży więcej niż dwóch przedmiotów. Algorytm jest tak skonstruowany, że co najwyżej możemy użyć kombinacji dwóch plecaków — nigdy nie będziemy mieli więcej niż dwa podplecaki. Możliwe jest natomiast utworzenie podplecaków w podplecakach.



Czy najlepsze rozwiążanie zawsze oznacza całkowite zapełnienie plecaka?

Nie. Wyobraź sobie, że oprócz wcześniej wymienionych przedmiotów do ukradzenia jest też brylant.

To bardzo duży klejnot — waży aż 3,5 kg i jest wart milion złotych, a więc o wiele więcej niż którykolwiek z pozostałych przedmiotów. Zdecydowanie powinieneś go wziąć! Jednak wówczas w plecaku zostaje jeszcze miejsce na pół kilograma ładunku i niczym go już nie zapełnimy.



ĆWICZENIE

9.2. Jedziesz na biwak. Masz plecak o obciążeniu 6 kg i możesz zabrać przedmioty z poniższej listy. Każdy z nich ma określoną wartość. Im ona wyższa, tym ważniejszy jest dany produkt.

- Woda, 3 kg, 10
- Książka, 1 kg, 3
- Jedzenie, 2 kg, 3
- Kurtka, 2 kg, 5
- Aparat, 1 kg, 6

Jaki jest optymalny zestaw przedmiotów do zabrania na wycieczkę?

Najdłuższa wspólna część łańcucha

W poprzedniej części rozdziału opisałem jeden problem, który można rozwiązać za pomocą programowania dynamicznego. Co należy zapamiętać?

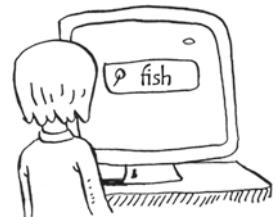
- Programowanie dynamiczne jest przydatne, gdy trzeba *zoptimali-zować coś, stosując się do określonych warunków*. W przypadku problemu plecaka naszym zadaniem była maksymalizacja wartości kradzionych towarów, a warunkiem — ograniczona pojemność plecaka.
- Za pomocą programowania dynamicznego można rozwiązywać problemy dające się podzielić na dyskretnie podproblemy, które nie są od siebie wzajemnie zależne.



Opracowanie rozwiązania z wykorzystaniem programowania dynamicznego może być trudne. Na tym skupimy się w tym podrozdziale. Oto kilka ogólnych porad.

- Każde rozwiązanie oparte na programowaniu dynamicznym wykorzystuje siatkę.
- Wartości w komórkach to zazwyczaj to, co próbujemy zoptymalizować. W przypadku zadania z plecakiem wartości te były cenami produktów.
- Każda komórka reprezentuje podproblem, więc zawsze należy zastanowić się, jak podzielić zadanie na podproblemy. W ten sposób łatwiej określić osie.

Przeanalizujemy jeszcze jeden przykład. Powiedzmy, że jesteśmy właścicielem portalu *dictionary.com*. Gdy w polu wyszukiwania ktoś wpisze słowo, program wyświetli jego definicję.



A kiedy słowo zostanie wpisane błędnie, program stara się odgadnąć, co użytkownik miał na myśli. Aleks szuka wyrazu *fish*, tylko przez przypadek wpisał *hish*. W słowniku nie ma takiego słowa, ale jest kilka podobnych.

Podobne do „hish”:

- *fish*
- *vista*

(To tylko przykład, więc ograniczymy się do dwóch słów, ale w prawdziwej aplikacji taka lista prawdopodobnie zawierałaby tysiące słów).

Aleks wpisał *hish*. Które słowo naprawdę miał na myśli: *fish* czy *vista*?

Przygotowanie siatki

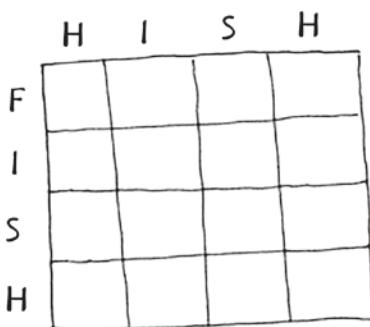
Jak wygląda siatka do tego problemu? Musimy odpowiedzieć na następujące pytania.

- Jakie są wartości komórek?
- Jak podzielić problem na części?
- Jakie są osie siatki?

W programowaniu dynamicznym chodzi o *maksymalizację* czegoś. W tym przypadku chcemy znaleźć najdłuższą wspólną część dwóch słów. Co mają wspólnego słowa *hish* i *fish*? A co wspólnego mają ze sobą słowa *hish* i *vista*? To właśnie chcemy obliczyć.

Pamiętaj, że tym, co chcesz zoptymalizować, są zazwyczaj wartości komórek. W tym przypadku wartościami najprawdopodobniej będą liczby oznaczające długość podłańcucha, który występuje w obu porównywanych łańcuchach.

Jak podzielić ten problem na części? Można porównywać podłańcuchy. Zamiast *hish* i *fish* najpierw można porównać *his* i *fis*. Każda komórka będzie zawierała wartość określającą długość najbliższego podłańcucha wspólnego dla obu porównywanych łańcuchów. Dodatkowo podsufa nam to podpowiedź, że osiąmi będą prawdopodobnie te dwa słowa. Zatem nasza siatka powinna wyglądać tak, jak poniżej.



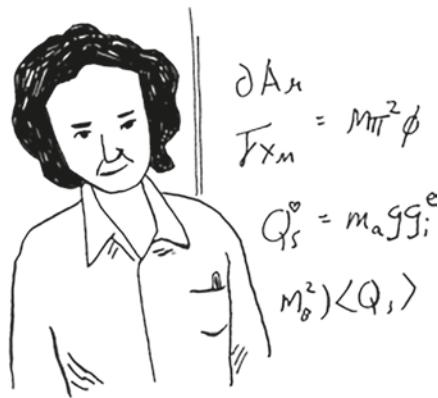
Nie przejmuj się, jeżeli to wszystko wydaje Ci się czarną magią. To trudne rzeczy i dlatego wziąłem się za nie dopiero pod koniec książki! Dalej zamieścilem jeszcze ćwiczenie, abyś mógł samodzielnie poćwiczyć programowanie dynamiczne.

Wypełnianie siatki

Wiesz już, jak powinna wyglądać siatka. Za pomocą jakiego wzoru obliczymy wartości wszystkich komórek? Możemy pójść na skróty, ponieważ już znamy rozwiązanie — *hish* i *fish* mają wspólny podłańcuch składający się z trzech liter — *ish*.

To jednak nie przybliża nas do wzoru. Informatycy czasami żartują sobie, że najlepiej użyć **algorytmu Feynmana**. Nazwa tego algorytmu pochodzi od nazwiska słynnego fizyka Richarda Feynmana, a oto jego opis.

1. Zapisz problem.
2. Myśl bardzo intensywnie.
3. Zapisz rozwiązanie problemu.



Informatycy to prawdziwi żartownisie!

Prawda jest jednak taka, że w tym przypadku nie da się w łatwy sposób utworzyć wzoru. Trzeba trochę poeksperymentować i wypróbować różne możliwości. Algorytmy nie zawsze są dokładnymi przepisami do realizacji. Mogą też opisywać ramowe rozwiązańia, na bazie których można formułować własne pomysły.

Spróbuj samodzielnie opracować rozwiązanie tego problemu. Dam Ci odpowiedź — część siatki będzie wyglądała tak, jak na poniższym rysunku.

	H	I	S	H
F	O	O		
I				
S			2	O
H				3

Jakie będą pozostałe wartości? Pamiętaj, że każda komórka reprezentuje wartość **podproblemu**. Dlaczego w komórce (3,3) znajduje się liczba 2? Dlaczego komórka (3,4) zawiera liczbę 0?

Najpierw samodzielnie spróbuj wymyślić ten wzór, a następnie przeczytaj rozwiązanie. Jeśli nawet nie udało Ci się wpaść na właściwy trop, dzięki moim objaśnieniom z pewnością wszystko zrozumiesz.

Rozwiązanie

Oto cała siatka.

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

A to wzór, przy użyciu którego wypełniłem komórki.

1. Jeśli litery

nie pasują,
wstawiane
jest zero

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

2. Jeśli litery pasują, to wstawiana jest wartość sąsiada z lewego górnego rogu powiększona o 1

A tak wygląda ten wzór w postaci pseudokodu.

```
if word_a[i] == word_b[j]: Litery pasują.
    cell[i][j] = cell[i-1][j-1] + 1
else: Litery nie pasują.
    cell[i][j] = 0
```

Oto siatka dla słów *hish* i *vista*.

	V	I	S	T	A
H	0	0	0	0	0
I	0	1	0	0	0
S	0	0	2	0	0
H	0	0	0	0	0

↑ Ostateczna odpowiedź ↑ To nie jest ostateczna odpowiedź

Drobna uwaga: w tym przypadku ostateczne rozwiązywanie może nie wypaść w ostatniej komórce! W przypadku problemu plecaka to ostatnia komórka zawsze zawierała rozwiązywanie. Natomiast przy szukaniu najdłuższego wspólnego podłańcucha rozwiązyaniem jest największa wartość z wszystkich komórek — nie musi ona znajdować się w ostatniej komórce.

Wróćmy do naszego pytania: „Który łańcuch ma więcej wspólnego z łańcuchem *hish*?”. Już wiemy, że *hish* i *fish* mają trzy wspólne litery, a *hish* i *vista* tylko dwie.

Zatem Aleks najprawdopodobniej chciał wpisać *fish*.

Najdłuższa wspólna podsekwencja

Powiedzmy, że Aleks przez przypadek wpisał *fosh*. Co miał na myśli: *fish* czy *fort*?

Sprawdźmy to za pomocą wzoru na obliczanie najdłuższego wspólnego podłańcucha.

F	O	S	H	
F	1	0	0	0
O	0	2	0	0
R	0	0	0	0
T	0	0	0	0

vs

F	O	S	H	
F	1	0	0	0
I	0	0	0	0
S	0	0	1	0
H	0	0	0	2

Wyniki są takie same: dwie litery! Jednak łańcuchowi *fosh* jest bliżej do *fish*.

$$\begin{array}{cccc} F & O & S & H \\ \downarrow & \downarrow & \downarrow & \\ F & I & S & H \end{array} = 3$$

$$\begin{array}{cccc} F & O & S & H \\ \downarrow & \downarrow & & \\ F & O & R & T \end{array} = 2$$

Porównujemy najdłuższy wspólny *podłańcuch*, ale naprawdę powinniśmy porównywać najdłuższą wspólną *podsekwencję*, czyli liczbę liter w sekwencji, które są wspólne dla obu słów. Jak obliczyć najdłuższą wspólną podsekwencję?

Oto część siatki dla wyrazów *fish* i *fosh*.

	F	O	S	H
F	1	1		
I	1			
S			1	2
H			2	

Potrafisz podać wzór do obliczenia tej siatki? Najdłuższa wspólna podsekwenca jest bardzo podobna do najdłuższego wspólnego podłańcucha i wzory w obu przypadkach także są bardzo do siebie podobne. Spróbuj wymyślić coś samodzielnie, a potem przeczytaj moje rozwiązańe.

Najdłuższa wspólna podsekwenca — rozwiązanie

Oto ostateczna wersja siatki.

	F	O	S	H
F	1 → 1 → 1 → 1			
O	↓ 1 → 2 → 2 → 2			
R	↓ 1 → 2 → 2 → 2			
T	↓ 1 → 2 → 2 → 2			

Najdłuższa wspólna podsekwenca = 2

vs

	F	O	S	H
F	1 → 1 → 1 → 1			
I	↓ 1 → 1 → 1 → 1			
S	↓ 1 → 1	2 → 2		
H	↓ 1 → 1	2	3	

Najdłuższa wspólna podsekwenca = 3

Poniżej przedstawiam wzór, za pomocą którego obliczyłem wartości dla wszystkich komórek.



A oto implementacja w postaci pseudokodu.

```

if word_a[i] == word_b[j]: ..... Litera są takie same.
    cell[i][j] = cell[i-1][j-1] + 1
else: ..... Litera są różne.
    cell[i][j] = max(cell[i-1][j], cell[i][j-1])

```

Uff! To był zdecydowanie najtrudniejszy z wszystkich rozdziałów w tej książce. Czy ktoś w ogóle używa programowania dynamicznego w praktyce? Tak.

- Biolodzy wyszukują najdłuższe wspólne sekwencje w celu identyfikacji podobieństw w łańcuchach DNA. W ten sposób potrafią stwierdzić podobieństwo pary zwierząt albo dwóch chorób. Przy użyciu tej techniki poszukiwane jest m.in. lekarstwo na stwardnienie rozsiane.
- Używałeś kiedyś narzędzia porównującego (np. `git diff`)? Służy ono do wyszukiwania różnic między dwoma plikami i działa właśnie z wykorzystaniem algorytmów programowania dynamicznego.
- Skoro mowa o podobieństwach łańcuchów, to **odległość Levenshteina** określa stopień podobieństwa dwóch łańcuchów — również przy użyciu

programowania dynamicznego. Miara ta znajduje szereg zastosowań, od modułów sprawdzania pisowni po systemy antyplagiatowe.

- Czy używałeś kiedykolwiek programu zawijającego wiersze tekstu, takiego jak np. Microsoft Word? Skąd taki program „wie”, w którym miejscu zrobić przeniesienie, aby długości linijek tekstu były takie same? Programowanie dynamiczne!

ĆWICZENIE

- 9.3.** Narysuj i wypełnij danymi siatkę do obliczania najdłuższego wspólnegołańcucha dla napisów *blue* i *clues*.

Powtórzenie

- Za pomocą technik programowania dynamicznego można wykonywać optymalizacje na podstawie określonych warunków.
- Programowanie dynamiczne można stosować, gdy dany problem da się podzielić na osobne podproblemy.
- Każde rozwiązanie oparte na programowaniu dynamicznym wykorzystuje siatkę.
- Wartości w komórkach to zazwyczaj dane, które użytkownik próbuje zoptymalizować.
- Każda komórka reprezentuje podproblem, więc należy się zastanowić, jak podzielić problem na podproblemy.
- Nie istnieje jedyny poprawny wzór na obliczanie rozwiązań w programowaniu dynamicznym.

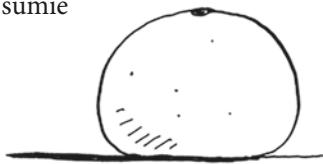


W tym rozdziale:

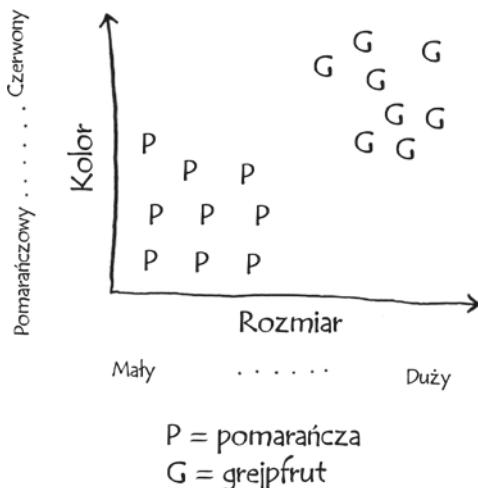
- dowiesz się, jak opracować system klasyfikacji przy użyciu algorytmu *k* najbliższych sąsiadów,
- poznasz pojęcie wyboru cech,
- dowiesz się, czym jest regresja, czyli przewidywanie liczb, np. jakie będą jutrzysze wartości akcji albo jak bardzo widzowi spodoba się film,
- poznasz różne zastosowania i ograniczenia algorytmu *k* najbliższych sąsiadów.

Klasyfikacja pomarańczy i grejpfrutów

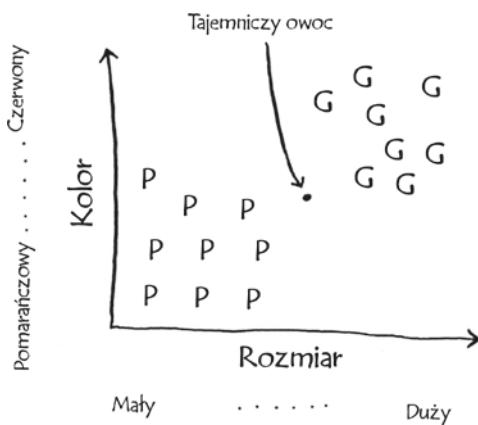
Spójrz na ten owoc. Czy to jest pomarańcza, czy grejpfrut? W sumie wiem, że grejpfruty są na ogół większe i bardziej czerwone.



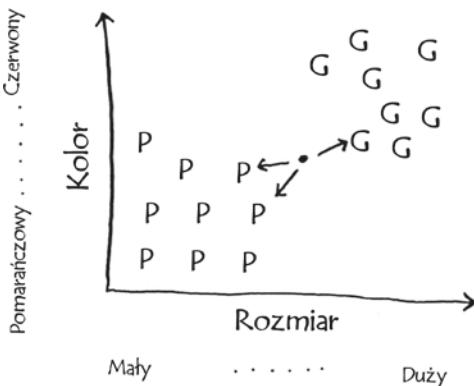
Zastanawiając się nad postawionym problemem, tworzę sobie w myślach graf.



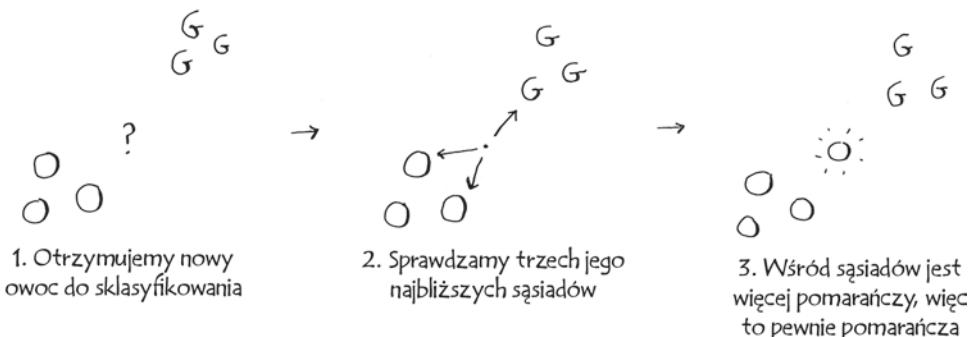
Na ogół większe i bardziej czerwone owoce są grejpfrutami. Ten owoc jest duży i czerwony, więc pewnie jest grejpfrutem. Ale co zrobić z takim owocem jak poniższy?



Jak sklasyfikować ten owoc? Jedną z możliwości jest przyjrzenie się sąsiednim owocom. Spójrzmy zatem na trzy najbliższe owoce.



W najbliższym sąsiedztwie jest więcej pomarańczy niż grejpfrutów, więc najprawdopodobniej to jest pomarańcza. Gratulacje: właśnie zastosowałeś do *klasyfikacji* algorytm *k najbliższych sąsiadów* (ang. *k-nearest neighbors* —KNN)! Zasada działania tego algorytmu jest bardzo prosta.

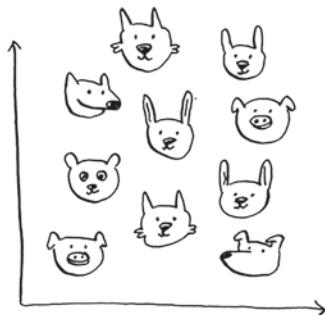


Mimo swojej prostoty algorytm KNN jest bardzo przydatny! Jeśli masz coś do zaklasyfikowania, możesz z niego skorzystać w pierwszej kolejności. Spójrzmy na bardziej realistyczny przykład.

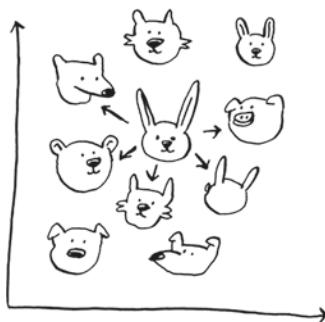
Budowa systemu rekomendacji

Wyobraź sobie, że jesteś właścicielem Netflixa i chcesz zbudować system rekomendacji filmów dla użytkowników. Gdyby spojrzeć na ten problem z dystansu, można zauważyc, że jest podobny do problemu z grejpfrutami.

Wszystkich użytkowników można przedstawić na grafie.



Miejsce na grafie jest wybierane na podstawie podobieństwa, a więc użytkownicy o podobnym guście są grupowani w pobliżu. Powiedzmy, że chcemy polecić filmy Patrycji. W związku z tym szukamy pięciu znajdujących się najbliżej niej innych użytkowników.



Podobny gust mają też Justyna, Janek, Lidia i Krysia. Można więc przypuszczać, że jeśli jakiś film spodobał się im, Patrycji też może się spodobać!

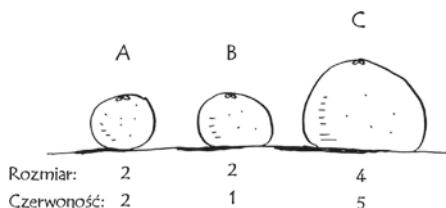
Z takim grafem budowa systemu rekomendacji jest już łatwa. Jeśli Justynie podoba się dany film, możemy go polecić Patrycji.



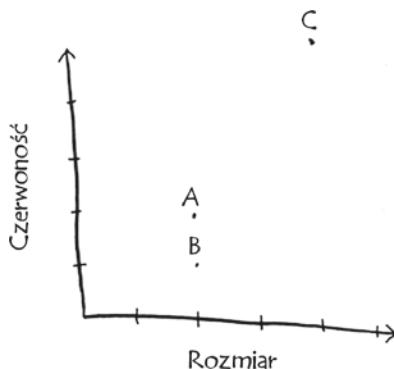
W tej układance brakuje jednak ważnego elementu. Na grafie rozmieściliśmy użytkowników wg podobieństwa gustów. Tylko jak określić to podobieństwo?

Wybór cech

W przykładzie dotyczącym grejpfrutów porównywaliśmy owoce na podstawie rozmiaru i stopnia czerwoności. Innymi słowy, porównywaliśmy takie cechy jak rozmiar i kolor. Teraz powiedzmy, że mamy trzy poniższe owoce i możemy określić ich cechy.



Przy użyciu posiadanych informacji te trzy owoce możemy przedstawić na grafie.



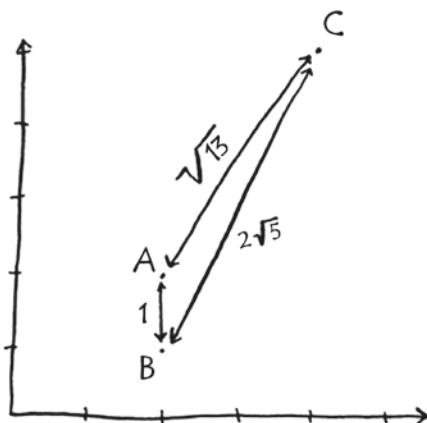
Na podstawie tego grafu możemy stwierdzić, że owoce A i B są do siebie podobne. Sprawdźmy, jak bardzo są sobie bliskie. Odległość między dwoma punktami można obliczyć za pomocą twierdzenia Pitagorasa.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Oto przykładowe obliczenia dystansu dzielącego punkty A i B.

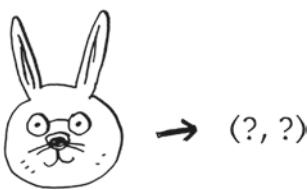
$$\begin{aligned} & \sqrt{(2-2)^2 + (2-1)^2} \\ &= \sqrt{0+1} \\ &= \sqrt{1} \\ &= 1 \end{aligned}$$

Punkty A i B dzieli odległość 1. Pozostałe odległości też można obliczyć.



Wyniki obliczeń potwierdzają to, co stwierdziliśmy „na oko”, tzn. że owoce A i B są do siebie podobne.

A teraz wyobraź sobie, że zamiast owoców chcesz porównywać użytkowników usługi Netflix. W tym celu musisz ich jakość przełożyć na graf. Możesz wykorzystać podobne rozwiązań jak w przypadku owoców, tzn. każdemu użytkownikowi przypisać współrzędne określające jego położenie.



Po rozmieszczeniu użytkowników na grafie można zacząć obliczanie dzielących ich odległości.

Oto, jak można przekonwertować każdego użytkownika na zestaw liczb. Gdy ktoś rejestruje się w usłudze Netflix, należy poprosić go o ocenienie kilku kategorii filmów. W ten sposób otrzymujemy zbiór ocen każdego użytkownika!

			
Patrycja			
Komedia	3	4	2
Akcja	4	3	5
Dramat	4	5	1
Horror	1	1	3
Romans	4	5	1

Patrycja i Justyna lubią romanse i nie znoszą horrorów. Morfeusz lubi filmy akcji, ale nie cierpi romansów (nawet strasznie się denerwuje, gdy dobry film akcji psują jakieś mdławie romantyczne sceny). Pamiętasz, jak w przykładzie z pomarańczami i grejpfrutami każdy owoc był reprezentowany przez dwie liczby? W tym przypadku natomiast każdy użytkownik będzie reprezentowany przez zestaw pięciu liczb.

$$\text{apple} \rightarrow (2, 2)$$

$$\text{rabbit} \rightarrow (3, 4, 4, 1, 4)$$

Matematyk powiedziałby, że teraz zamiast w dwóch wymiarach obliczamy odległość w *pięciu* wymiarach. Mimo to wzór pozostaje taki sam.

$$\sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (c_1 - c_2)^2 + (d_1 - d_2)^2 + (e_1 - e_2)^2}$$

Po prostu zamiast zestawu dwóch liczb teraz mamy zestaw pięciu liczb.

Wzór na obliczanie odległości jest elastyczny: można nawet utworzyć zestaw miliona liczb i do obliczenia dystansu posłużyć się tym samym starym wzorem. Może się zastanawiasz: „Co znaczy słowo *dystans*, gdy jest pięć liczb?”. Informuję o tym, jak bardzo podobne są do siebie te zestawy liczb.

$$\begin{aligned} & \sqrt{(3-4)^2 + (4-4)^2 + (4-5)^2 + (1-1)^2 + (4-5)^2} \\ &= \sqrt{1+1+1+0+1} \\ &= \sqrt{4} \\ &= 2 \end{aligned}$$

Oto dystans dzielący Patrycję i Justynę.

Patrycja i Justyna mają bardzo podobne gusty. A jaka jest różnica między Patrycją i Morfeuszem? Oblicz odległość między nimi, zanim przejdziesz dalej.

Ciekawe, czy udało Ci się poprawnie wykonać obliczenia. Patrycję i Morfeusza dzieli dystans 24. Z tego wynika, że pod względem gustu filmowego Patrycji jest o wiele bliżej do Justyny niż do Morfeusza.

Świetnie! Teraz polecanie filmów Patrycji jest już łatwe — jeśli Justynie podoba się dany film, można go polecić także Patrycji i odwrotnie. Właśnie opracowaliśmy system rekommendacji!

Jeśli jesteś klientem Netfliksa, cały czas namawiają Cię, abyś oceniał więcej filmów. Im więcej ocen podasz, tym trafniejsze będziesz otrzymywać rekommendacje. Teraz już wiesz czemu. Im więcej filmów ocenisz, tym skuteczniej Netflix jest w stanie znaleźć użytkowników o guście podobnym do Twojego.

ĆWICZENIA

- 10.1.** W przykładzie dotyczącym Netfliksa dystans dzielący dwóch użytkowników obliczaliśmy za pomocą specjalnego wzoru. Jednak nie wszyscy użytkownicy oceniają filmy w taki sam sposób. Powiedzmy, że mamy dwóch użytkowników, nazwijmy ich Yogi i Pinky, którzy mają taki sam gust filmowy. Mimo to Yogi wszystkim filmom, które mu się podobają, przyznaje piątkę, a Pinky jest bardziej wybredny i najwyższą ocenę przyznaje tylko naprawdę — jego zdaniem — najlepszym filmom. Choć

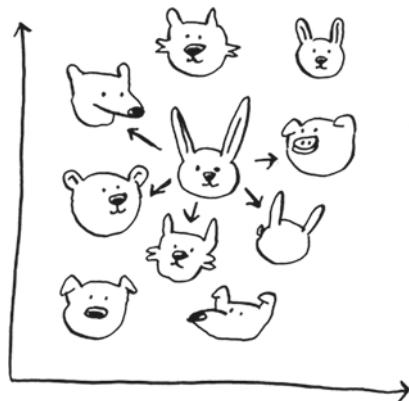
panowie mają podobne gusty, według naszego algorytmu obliczania dystansu nie są sąsiadami. Co zrobisz, by uwzględnić w obliczeniach także te odmienne strategie oceniania?

- 10.2.** Powiedzmy, że Netflix wyznacza grupę „liczących się użytkowników”. Zaliczają się do nich np. Quentin Tarantino oraz Wes Anderson i ich oceny liczą się znacznie bardziej niż zwykłych użytkowników. Jak zmodyfikowałbyś system rekomendacji, aby bardziej zdecydowanie uwzględniał opinie takich osób?

Regresja

Powiedzmy, że chcemy nie tylko polecać filmy, ale dodatkowo np. zgadywać, jak Patrycja oceni dany film. Weźmy pięć najbliższych jej osób.

Przy okazji, choć ciągle piszę o pięciu najbliższych osobach, nie ma w tej liczbie nic niezwykłego. Równie dobrze mogłyby to być dwie najbliższe osoby albo dziesięć czy dziesięć tysięcy najbliższych osób. Dlatego właśnie algorytm nazywa się k najbliższych sąsiadów, a nie 5 najbliższych sąsiadów!



Powiedzmy, że chcemy odgadnąć ocenę filmu *Pitch Perfect*. Jak ocenili go Justyna, Janek, Lidia i Krysia?

Justyna: 5

Janek: 4

Stefan: 4

Lidia: 5

Krysia: 3

Moglibyśmy policzyć średnią tych ocen, która wynosi 4.2 gwiazdki. Nazywa się to **regresją**. Algorytm KNN ma dwa główne zastosowania, należą do nich klasyfikacja i regresja.

- Klasyfikacja to przyporządkowywanie elementów do grup.
- Regresja to przewidywanie odpowiedzi (np. liczb).

Regresja to metoda, która przydaje się w wielu sytuacjach. Wyobraź sobie, że prowadzisz małą piekarnię w Bydgoszczy i codziennie pieczesz świeży chleb. Próbujesz przewidzieć, ile bochenków powinieneś upiec dzisiaj. Masz do dyspozycji zestaw cech.

- Pogoda w skali od 1 do 5 (1 = zła, 5 = piękna).
- Weekend lub święto (1 oznacza weekend lub święto, 0 inny dzień).
- Czy trwają jakieś rozgrywki (1 jeśli tak, 0 jeśli nie).

Wiesz już, ile bochenków chleba sprzedawałeś w przeszłości, gdy zestawy cech były inne.



A. $(5, 1, 0) = 300$
bochenków

B. $(3, 1, 1) = 225$
bochenków

C. $(1, 1, 0) = 75$
bochenków

D. $(4, 0, 1) = 200$
bochenków

E. $(4, 0, 0) = 150$
bochenków

F. $(2, 0, 0) = 50$
bochenków

Dzisiaj mamy weekend i dobrą pogodę. Opierając się na przedstawionych wcześniej danych, jak myślisz, ile bochenków uda się dzisiaj sprzedać? Wykorzystamy algorytm KNN, parametrowi K nadając wartość 4. Najpierw poszukamy czterech najbliższych sąsiadów dla naszego punktu.

$$(4, 1, O) = ?$$

Oto odległości. Najbliższe są punkty A, B, D i E.

A. 1 ←

B. 2 ←

C. 9

D. 2 ←

E. 1 ←

F. 5

Obliczamy średnią liczbę bochenków sprzedanych w tych dniach. Otrzymujemy wartość 218,75 i tyle właśnie powinniśmy dzisiaj upiec!

Podobieństwo kosinusowe

Do tej pory do porównywania dystansu dzielącego dwóch użytkowników używaliśmy wzoru na odległość. Czy to najlepszy wzór tego typu? W praktyce powszechnie wykorzystuje się też **podobieństwo kosinusowe**. Powiedzmy, że dwaj użytkownicy są do siebie podobni, ale jeden z nich jest bardziej wstrzemięźliwy w wydawaniu opinii. Obu bardzo podobał się film *Amar Akbar Anthony* Manmohana Desaia. Paweł ocenił dzieło na pięć gwiazdek, a Roman dał mu tylko cztery gwiazdki. Jeśli użyjemy wzoru na odległość, ci dwaj użytkownicy mogą nie być sąsiadami, mimo że mają podobne gusty.

Podobieństwo kosinusowe nie mierzy odległości między dwoma wektorami, tylko jest miarą porównania kątów dwóch wektorów i lepiej sprawdza się w takich przypadkach jak ten. W książce tej nie ma miejsca na szczegółowy opis tej miary, ale poszukaj o niej więcej informacji, jeśli zamierzasz wykorzystywać algorytm KNN!

Wybieranie odpowiednich cech



Aby wiedzieć, co polecamy użytkownikom, prosiliśmy ich o ocenianie kategorii filmów. A co by było, gdybyśmy zamiast tego dawali im do oceny zdjęcie kotów? Wówczas wiedzielibyśmy, którzy użytkownicy w podobny sposób oceniają te obrazy. Taki system rekomendacji pewnie byłby mniej skuteczny, ponieważ oceniane „cechy” nie miałyby wiele wspólnego z gustem kinowym!

Albo wyobraź sobie, że prosisz użytkowników o oceny filmów, aby przedstawić im rekomendacje — ale do oceny podsuwasz tylko *Toy Story*, *Toy Story 2* i *Toy Story 3*. W ten sposób również niewiele dowiesz się o guście widzów!

Ważnym aspektem pracy z algorytmem KNN jest wybór odpowiednich cech do porównywania. W naszym przypadku odpowiednie cechy to takie, które:

- bezpośrednio korelują z filmami, jakie próbujemy polecać,
- nie są w żaden sposób nastawione (jeśli np. będziemy prosić użytkowników o ocenianie samych komedii, niewiele dowiemy się na temat ich gustu w odniesieniu do filmów akcji).

Czy uważasz, że polecanie filmów na podstawie ich ratingów to dobry pomysł? Może *Prawo ulicy* ocenilem wyżej niż *House Hunters*, ale w rzeczywistości częściej oglądam ten drugi program? Jak byś ulepszył ten system rekomendacji Netfliksa?

Wracamy do piekarni: potrafisz podać dwie niewłaściwe i dwie odpowiednie cechy, których można by użyć w tym przypadku? Może trzeba upiec więcej chleba po opublikowaniu reklamy w gazecie. A może więcej bochenków trzeba piec w poniedziałki?

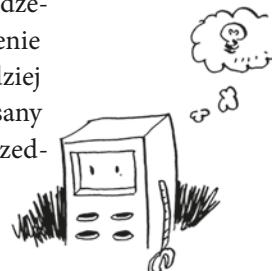
W kwestii wyboru odpowiednich cech nie ma jedynej poprawnej odpowiedzi. Zawsze trzeba dokładnie rozważyć wszystkie okoliczności.

ĆWICZENIE

10.3. Netflix ma miliony użytkowników. Opisany w przykładzie system rekomendacji opierał się na analizie cech pięciu najbliższych sąsiadów. Czy myślisz, że to za dużo? A może za mało?

Wprowadzenie do uczenia maszynowego

KNN to bardzo przydatny algorytm, który jednocześnie stanowi wprowadzenie do magicznego świata uczenia maszynowego! Najkrócej mówiąc, uczenie maszynowe to technika pozwalająca sprawić, że komputer stanie się bardziej inteligentny. Jeden przykład jej zastosowania już pokazałem — jest to opisany w poprzednim podrozdziale system rekomendacji. Teraz chciałbym przedstawić jeszcze kilka innych przykładów.



Optyczne rozpoznawanie znaków

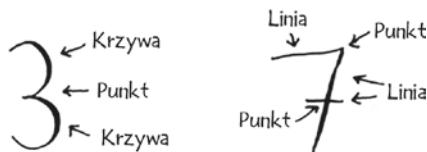
Optyczne rozpoznawanie znaków (ang. *optical character recognition*) to technologia, przy której można wykonać fotografię tekstu, a następnie wprowadzić ją do komputera w celu odczytania tego tekstu. Za pomocą tej technologii firma Google digitalizuje książki. Jak to działa? Weźmy np. poniższą liczbę.

7

Jak z wykorzystaniem automatu rozpoznać, co to za liczba? Można do tego celu użyć algorytmu KNN.

1. Przejrzyj dużą liczbę obrazów liczb i rozpoznaj charakterystyczne cechy każdej z nich.
2. Kiedy otrzymasz nowy obraz, pobierz jego cechy i sprawdź, jakich ma najbliższych sąsiadów!

To taki sam rodzaj problemu jak rozróżnianie pomarańczy i grejpfrutów. Na ogół algorytmy OCR mierzą linie, punkty i krzywe.



Gdy dostaniemy do rozpoznania kolejny znak, będziemy mogli znaleźć na nim te same cechy.

Oczywiście proces pobierania cech w technologii OCR jest znacznie bardziej skomplikowany niż w przykładzie z owocami. Należy jednak wiedzieć, że nawet najbardziej skomplikowane technologie zbudowano w oparciu o proste pomysły, takie jak właśnie KNN. Na tej samej podstawie można opracować systemy rozpoznawania mowy albo twarzy. Gdy ktoś wysyła zdjęcia na Facebook, portal czasami automatycznie oznacza widoczne na nim osoby. To doskonały przykład praktycznego wykorzystania uczenia maszynowego!

Pierwszy etap procesu OCR, w ramach którego przegląda się obrazy liczb i określa ich cechy, nazywa się **szkoleniem**. Etap ten jest wbudowany w większość algorytmów uczenia maszynowego. Zanim komputer wykona zadanie, trzeba go najpierw tego nauczyć. Następny przykład dotyczy filtrów spamu, które również mają wbudowany mechanizm szkolenia.

Budowa filtra spamu

Filtry spamu wykorzystują jeszcze inny prosty rodzaj algorytmu o nazwie **naiwny klasyfikator bayesowski** (ang. *naive Bayes classifier*). Najpierw szkoli się taki klasyfikator za pomocą próbki danych.

Temat	Spam?
„Zresetuj swoje hasło”	Nie spam
„Wygrałeś milion dolarów”	Spam
„Wyślij mi swoje hasło”	Spam
„Nigeryjski książę przesyła ci milion dolarów”	Spam
„Wszystkiego najlepszego”	Nie spam

Powiedzmy, że otrzymujemy e-mail o temacie: „Już teraz odbierz swój milion dolarów!”. Czy to jest spam? Można podzielić to zdanie na słowa, a następnie sprawdzić, z jakim prawdopodobieństwem każde z nich może wystąpić w śmieciowej wiadomości e-mail. W tym prostym przykładzie słowo *milion* występuje wyłącznie w spamie. W efekcie naiwny klasyfikator bayesowski stwierdza, że wiadomość o takim tytule to najprawdopodobniej spam. Ma on więc podobne zastosowanie jak algorytm KNN.

Za pomocą naiwnego klasyfikatora bayesowskiego można np. klasyfikować owoce — mamy owoc, który jest duży i czerwony. Jakie jest prawdopodobieństwo, że to grejpfrut? To kolejny prosty algorytm charakteryzujący się wysoką efektywnością. Kochamy takie algorytmy!



Przewidywanie cen akcji

Oto zadanie, które trudno wykonać przy użyciu uczenia maszynowego; jest to przewidywanie, czy ceny akcji na giełdzie wzrosną, czy zmaleją. Jakie cechy będą odpowiednie w takim przypadku? Powiedzmy, że uznajemy, iż skoro wczoraj ceny akcji poszły w górę, pójdu w górę i dzisiaj. Czy to dobra cecha? A może powinniśmy przyjąć, że ceny akcji zawsze spadają w maju? Czy to jest właściwe założenie? Nie da się ze stuprocentową pewnością przewidzieć przyszłych wartości na podstawie danych z przeszłości. Przewidywanie jest trudne, a przy tak dużej liczbie zmiennych wręcz niemożliwe.

Powtórzenie

Mam nadzieję, że masz już ogólne pojęcie o tym, co można zdziałać przy użyciu algorytmu KNN i uczenia maszynowego! Uczenie maszynowe to bardzo szeroka dziedzina, którą warto zgłębić szczegółowo, jeśli jesteś tym zainteresowany.

- Algorytm KNN jest wykorzystywany w klasyfikacji i regresji, a jego działanie polega na analizie k najbliższych sąsiadów.
- Klasyfikacja = przyporządkowywanie do grup.

- Regresja = przewidywanie odpowiedzi (np. liczby).
- Pobieranie cech to czynność polegająca na konwersji elementu (np. owocu albo użytkownika) na listę liczb, które można porównywać.
- Wybór odpowiednich cech jest ważnym czynnikiem mającym wpływ na powodzenie algorytmu KNN.

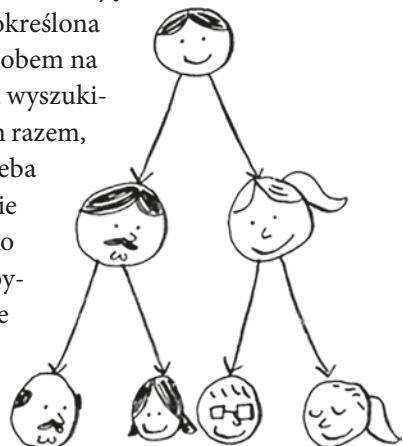


W tym rozdziale:

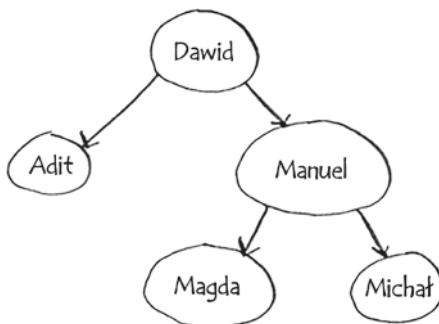
- zamieszczam zwięzły przegląd 10 algorytmów, które nie zostały opisane w tej książce, a które też są przydatne,
- dodaję porady, co warto przeczytać dalej w zależności od zainteresowań.

Drzewa

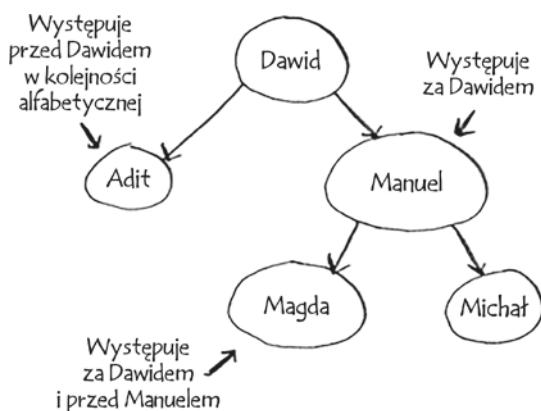
Wracamy do przykładu z drzewem binarnym. Gdy użytkownik loguje się na Facebooku, wewnętrzne mechanizmy portalu przeszukują wielką tablicę, aby sprawdzić, czy znajduje się w niej określona nazwa użytkownika. Stwierdziłem, że najszybszym sposobem na przeszukanie tej struktury jest skorzystanie z algorytmu wyszukiwania binarnego. Jednak jest pewien problem: za każdym razem, gdy zarejestruje się nowy użytkownik, jego nazwę trzeba wstawić do tablicy. Następnie konieczne jest posortowanie tej struktury, ponieważ wyszukiwanie binarne działa tylko z tablicami posortowanymi. Czy nie byłoby fajnie, gdybyśmy mogli wstawiać nazwy użytkowników od razu we właściwe miejsce, aby potem nie trzeba było sortować całej tablicy? Jest to możliwe w strukturze danych zwanej **binarnym drzewem poszukiwań** (ang. *binary search tree*).



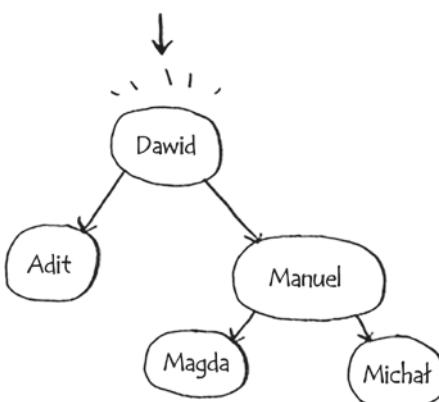
Oto binarne drzewo poszukiwań.



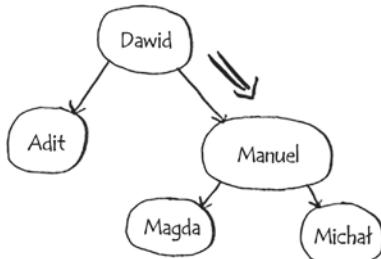
Dla każdego węzła węzły znajdujące się po jego lewej stronie mają mniejszą wartość, a węzły znajdujące się po prawej stronie mają większą wartość.



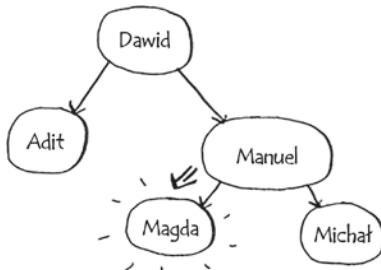
Powiedzmy, że szukamy Magdy. Poszukiwania zaczynamy od węzła głównego.



Magda jest za Dawidem, więc idziemy w prawo.



Magda jest przed Manuelem, więc teraz idziemy w lewo.



Znaleźliśmy Magdę! To prawie jak wyszukiwanie binarne! Znalezienie elementu w binarnym drzewie poszukiwań średnio zajmuje $O(\log n)$ czasu i $O(n)$ w najgorszym przypadku. Przeszukiwanie posortowanej tablicy trwa $O(\log n)$ w najgorszym przypadku, więc można pomyśleć, że posortowana tablica jest lepsza. Jednak binarne drzewo poszukiwań znacznie przewyższa tablicę pod względem średniej szybkości wstawiania i usuwania elementów.

	Tablica	Binarne drzewo poszukiwań
Szukanie	$O(\log n)$	$O(\log n)$
Wstawianie	$O(n)$	$O(\log n)$
Usuwanie	$O(n)$	$O(\log n)$

Binarne drzewa poszukiwań mają też wady, np. nie zapewniają swobodnego dostępu do elementów. Nie obsługują operacji w rodzaju: „Daj mi piąty element z tego drzewa”. Ponadto podane czasy to wartości średnie, które można

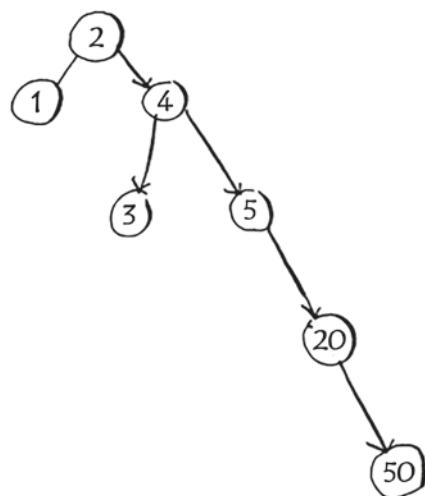
uzyskać w drzewach zrównoważonych. Powiedzmy, że mamy niezrównoważone drzewo binarne podobne do tego obok.

Widzisz, jak drzewo chyli się na prawą stronę? Przeszukiwanie takiej struktury nie będzie efektywne, ponieważ nie została zrównoważona. Istnieją specjalne drzewa binarne, które same się równoważą. Jednym z nich jest tzw. drzewo czerwono-czarne.

Do czego zatem używa się binarnych drzew poszukiwań? Specjalny rodzaj tych drzew o nazwie **B-drzewo** jest powszechnie używany do przechowywania danych w bazach danych.

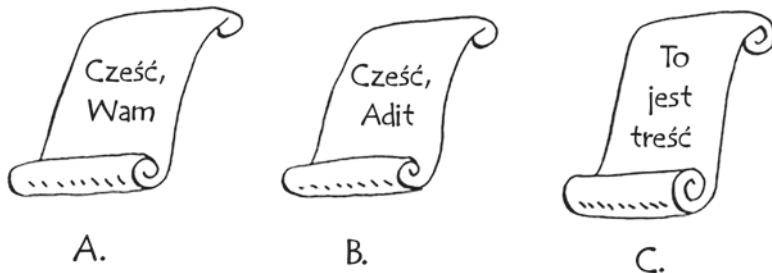
Jeśli interesują Cię bazy danych lub bardziej zaawansowane struktury danych, poszukaj informacji na poniższe tematy:

- B-drzewa,
- drzewa czerwono-czarne,
- sterty,
- drzewa rozchylane.



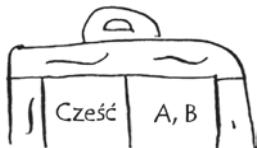
Odwrócone indeksy

Oto bardzo uproszczony opis działania wyszukiwarki internetowej. Powiedzmy, że mamy trzy następujące proste strony internetowe.



Z treści tych stron utworzymy tablicę skrótów.

Kluczami w tej tablicy są słowa, a wartości wskazują, na której stronie dane słowo się znajduje. Teraz powiedzmy, że użytkownik szuka słowa *cześć*. Sprawdźmy, na których stronach ono występuje.



Aha, to słowo występuje na stronach A i B.

Przedstawiamy zatem te strony w wynikach wyszukiwania. A teraz wyobraź sobie, że użytkownik szuka słowa *wam*. Wiemy, że to słowo występuje tylko na stronie A. Łatwe,

prawda? Struktura danych wiążąca słowa z miejscami ich występowania jest bardzo przydatna i nazywa się **odwróconym indeksem** (ang. *inverted index*). Najczęściej znajduje zastosowanie w wyszukiwarkach internetowych, więc jeśli interesują Cię te technologie, warto zacząć ich zgłębianie właśnie od tych struktur.

A simple line drawing of a backpack with a front pocket. Inside the pocket is a small table with four rows and two columns. The first row has 'Cześć' in the left column and 'A, B' in the right column. The second row has 'Adit' in the left column and 'B' in the right column. The third row has 'jest' in the left column and 'C' in the right column. The fourth row has 'treść' in the left column and 'C' in the right column.

Transformata Fouriera

Transformata Fouriera to jeden z najrzadszych algorytmów — błyskotliwy, elegancki i znajdujący liczne zastosowania. Najlepsze porównanie tego algorytmu znajduje się w portalu Better Explained (świetny serwis z prostymi objaśnieniami zagadnień matematycznych). Oto ono: gdyby transformatę Fouriera porównać do koktajlu, dzięki niej poznalibyśmy składniki tej mikstury¹. Albo inaczej mówiąc, gdyby wziąć piosenkę, transformata podzieliłaby ją na poszczególne częstotliwości.

Może się zdziwiś, ale ta prosta koncepcja ma wiele zastosowań. Przykładowo możliwość podzielenia piosenki na częstotliwości pozwala wybrać z niej te, które nas najbardziej interesują. Możemy np. podkręcić basy i wyciszyć wysokie dźwięki. Transformata Fouriera jest doskonałym narzędziem do przetwarzania sygnałów. Używa się jej też do kompresji muzyki. Najpierw dzieli się plik muzyczny na poszczególne nuty składowe. Następnie transformata Fouriera wskazuje, jak duże jest znaczenie każdej nuty w całym utworze, zatem można pozbyć się tych nut, które są nieistotne. Tak właśnie działa format MP3!

1. Kalid, „An Interactive Giude to the Fourier Transform”, Better Explained, <http://mng.bx/874X>

Muzyka to nie jedyny rodzaj sygnału cyfrowego. Innym formatem kompresji jest JPG, który działa w bardzo podobny sposób. Za pomocą transformaty Fouriera przewiduje się trzęsienia ziemi i analizuje DNA.

Przy jej użyciu można nawet utworzyć taką aplikację jak Shazam, która odgaduje, jaki utwór jest odtwarzany. Transformata Fouriera ma wiele zastosowań i jest duża szansa, że kiedyś się z nią zetkniesz!

Algorytmy równoległe

Trzy następne tematy dotyczą skalowalności i pracy z dużymi ilościami danych. Kiedyś szybkość komputerów rosła lawinowo, więc jeśli ktoś chciał, aby jakiś algorytm działał szybciej, wystarczyło poczekać kilka miesięcy, aż pojawią się szybsze komputery. Obecnie jednak zbliżamy się do końca tej ery. Aktualnie w laptopach i komputerach stacjonarnych montuje się procesory zawierające coraz więcej rdzeni. Aby przyspieszyć wykonywanie algorytmów, należy je tak projektować, żeby mogły być wykonywane równocześnie przez wszystkie rdzenie procesora!

Oto prosty przykład. Najlepszy wynik, jaki można uzyskać w przypadku algorytmu sortowania, to mniej więcej $O(n \log n)$. Powszechnie wiadomo, że nie da się posortować tablicy w czasie $O(n)$ — *chyba że używa się algorytmu równoległego!* Istnieje równoległa wersja algorytmu szybkiego sortowania, która sortuje tablice w czasie $O(n)$.

Projektowanie algorytmów równoległych jest trudne. Niełatwne jest też zapewnienie poprawności ich działania oraz określenie, jak dużo dzięki nim można zyskać na prędkości. Jedno jest jednak pewne — zyski na czasie nie są liniowe. Jeśli więc mamy w laptopie dwa rdzenie zamiast jednego, to prawie nigdy nie uda się sprawić, że algorytm magicznie zacznie być wykonywany dwa razy szybciej. Jest kilka powodów takiego stany rzeczy.

- **Narzut związany z zarządzaniem równoległymi operacjami** — powiedzmy, że chcemy posortować tablicę 1000 elementów. Jak podzielić takie zadanie między dwa rdzenie? Czy każdemu z nich dać po 500 elementów do posortowania, a następnie połączyć dwie posortowane tablice? Scalanie tablic też trwa.
- **Równoważenie obciążenia** — powiedzmy, że mamy do wykonania 10 zadań, więc każdemu rdzeniowi przydzielamy po 5. Rdzeń A otrzymuje łatwe zadania, więc kończy pracę w 10 sekund, podczas gdy rdzeń B otrzymuje

trudne zadania, przez co kończy pracę po minucie. W efekcie rdzeń A przez 50 sekund pozostawał bezczynny, a rdzeń B wykonał prawie całą pracę! Jak rozdzielić zadania w taki sposób, aby oba rdzenie miały mniej więcej tyle samo pracy do wykonania?

Jeśli interesujesz się teoretycznymi rozważaniami na temat wydajności i skalowalności, algorytmy równoległe to jest to!

MapReduce

W ostatnich czasach coraz większą popularność zdobywa specjalny rodzaj algorytmu równoległego o nazwie **algorytm rozproszony**. Algorytm równoległy dobrze się sprawdza, gdy np. w laptopie trzeba wykorzystać dwa lub cztery rdzenie. Co jednak zrobić, gdy potrzebnych jest kilkaset rdzeni? Wówczas można napisać algorytm wykonywany na wielu komputerach. MapReduce jest właśnie popularnym algorytmem rozproszonym, z którego można korzystać z pomocą otwartego narzędzia Apache Hadoop.

Do czego nadają się algorytmy rozproszone

Powiedzmy, że mamy tabelę z miliardami albo trylionami wierszy i chcemy na niej wykonać skomplikowane zapytanie SQL. Nie możemy użyć bazy MySQL, ponieważ ma trudności już przy kilku miliardach wierszy. Możemy natomiast skorzystać z MapReduce przy użyciu Hadoop!

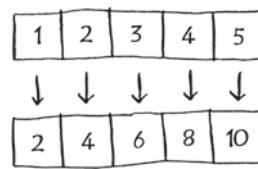
Albo wyobraź sobie, że musimy przetworzyć długą listę zadań. Wykonanie każdego z nich zajmuje 10 sekund i mamy ich aż 10 milionów. Gdybyśmy mieli do dyspozycji tylko jeden komputer, wykonanie wszystkich zadań zajęłoby nam miesiące. A gdybyśmy mogli użyć 100 komputerów, skrócilibyśmy czas wykonywania do kilku dni.

Algorytmy rozproszone są doskonałym wyborem, gdy trzeba wykonać dużo pracy w krótkim czasie. Algorytm MapReduce jest oparty na dwóch prostych filarach, czyli funkcjach `map` i `reduce`.

Funkcja map

Funkcja `map` jest bardzo prosta: pobiera tablicę i wykonuje tę samą funkcję na każdym elemencie tej tablicy. Poniżej np. podwijamy wartość każdego elementu tablicy.

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = map(lambda x: 2 * x, arr1)
[2, 4, 6, 8, 10]
```



Teraz tablica arr2 zawiera wartości [2, 4, 6, 8, 10] — każdy element tablicy arr1 został podwojony! Mnożenie wartości elementów przez dwa jest szybkie. Wyobraź sobie jednak, że w zamian stosujesz funkcję, której wykonywanie trwa długo. Spójrz na poniższy pseudokod:

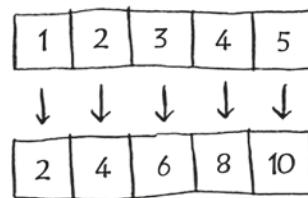
```
>>> arr1 = # lista adresów URL
>>> arr2 = map(download_page, arr1)
```

Mamy listę adresów URL i chcemy pobrać każdą stronę, aby zapisać jej treść w tablicy arr2. Wykonanie tych czynności dla każdego adresu może zająć kilka sekund. Gdybyśmy mieli 1000 adresów URL, wykonywanie zadania zajęłoby kilka godzin!

Czy nie byłoby wspaniale mieć 100 komputerów, na których funkcja `map` równomiernie rozkładałaby pracę do wykonania? Wówczas pobieralibyśmy 100 stron na raz, więc praca szłaby znacznie szybciej! Takie jest właśnie założenie, jeśli chodzi o sposób działania funkcji `map` w MapReduce.

Funkcja reduce

Niektórzy mają problem ze zrozumieniem roli funkcji `reduce`. Polega ona na tym, że celem całej operacji jest „redukacja” listy elementów do postaci jednego elementu. Przy użyciu funkcji `map` zamieniamy jedną tablicę na inną.

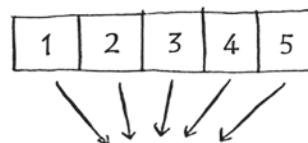


Natomiast funkcja `reduce` zamienia tablicę na pojedynczy element.

Oto przykład.

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
```

15



15

W tym przypadku sumujemy wszystkie elementy tablicy: $1 + 2 + 3 + 4 + 5 = 15$! Nie opisuję bardziej szczegółowo tej funkcji, ponieważ w internecie łatwo można znaleźć wiele samouczków.

MapReduce na podstawie tych dwóch prostych koncepcji wykonuje zapytania dotyczące danych rozproszonych na różnych komputerach. Gdy zbiór danych jest duży (miliardy wierszy), MapReduce potrafi znaleźć odpowiedź w kilka minut w przypadkach, w których zwykłe bazy danych mogłyby potrzebować godzin.

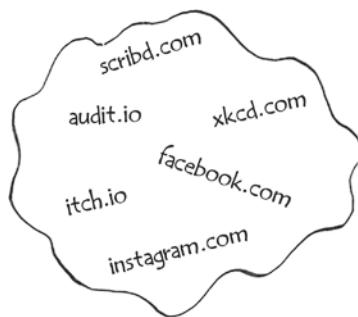
Filtry Bloom'a i HyperLogLog

Wyobraź sobie, że jesteś właścicielem portalu Reddit. Gdy ktoś publikuje odnośnik, sprawdzasz, czy ktoś inny wcześniej już go nie opublikował. Artykuły, które nie były do tej pory publikowane, są najcenniejsze. Dlatego musimy dowiedzieć się, jaki jest status publikowanego właśnie odnośnika.

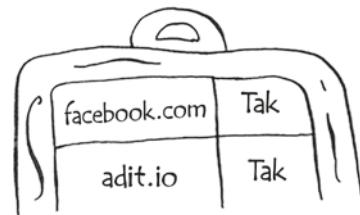
Albo wyobraź sobie, że jesteś portalem Google i przeszukujesz strony internetowe. Chcesz przeglądać tylko te strony, których wcześniej nie przejrzałeś. W takim przypadku musisz jakoś sprawdzić, czy dana strona była już przez Ciebie przeglądana, czy nie.

Albo pomyśl sobie, że jesteś właścicielem usługi skracania adresów bit.ly. Chcesz zablokować możliwość kierowania użytkowników do szkodliwych stron. Dlatego przygotowałeś zbiór adresów URL uważanych za szkodliwe i teraz za każdym razem musisz sprawdzać, czy adres, na który przekierujesz użytkownika, nie znajduje się w tym zbiorze.

Wszystkie te przykłady łączy ten sam problem — występowanie bardzo dużego zbioru danych.



Mamy nowy element i chcemy dowiedzieć się, czy należy do zbioru. Zadanie to możemy wykonać szybko przy użyciu tablicy skrótów. Powiedzmy np., że Google utrzymuje wielką tablicę skrótów, w której kluczami są przejrzone strony. Chcemy się dowiedzieć, czy już przejrzaliśmy serwis *adit.io*, więc szukamy go w tablicy.



adit.io → Tak

Stwierdzamy, że w tej tablicy znajduje się klucz *adit.io*, więc ta strona została już przeszukana. Średni czas wyszukiwania w tablicach skrótów wynosi O(1). Klucz *adit.io* jest w tablicy, co oznacza, że ta strona jest już przejrzana. Dowiedzieliśmy się o tym w stałym czasie. Całkiem nieźle.

Sąk tylko w tym, że ta tablica musi być *gigantyczna*. Google indeksuje tryliony stron internetowych. Aby w tablicy skrótów zapisać adresy wszystkich zaindeksowanych przez Google stron, trzeba by mieć bardzo dużo miejsca. Podobny problem mają Reddit i *bit.ly*. Gdy w grę wchodzą bardzo duże ilości danych, trzeba wykazać się pomysłowością!

Filtry Blooma

Rozwiązaniem omawianego problemu jest użycie filtrów Blooma. Te **probabilistyczne struktury danych** zwracają odpowiedź, która może być niepoprawna, ale prawdopodobnie jest prawidłowa. Zamiast tablicy skrótów możemy spytać filtr Blooma, czy już przejrzaliśmy stronę pod danym adresem. Tablica zwróciłaby dokładną odpowiedź, a filtr zwróci prawdopodobnie prawidłową odpowiedź.

- Możliwe są fałszywe potwierdzenia. Google może stwierdzić: „Już byłes na tej stronie”, nawet jeśli to nieprawda.
- Fałszywe zaprzeczenia są niemożliwe. Jeśli filtr Blooma stwierdzi: „Nie przejrzałeś jeszcze tej strony”, tzn. że na pewno jeszcze jej nie przejrzałeś.

Wielką zaletą filtrów Blooma jest to, że zajmują bardzo mało miejsca. Tablica skrótów musi przechowywać każdy adres URL odwiedzony przez Google, a filtr Blooma nie musi tego robić. Struktury te są doskonałym wyborem, gdy wyniki nie muszą być dokładne, jak we wszystkich tych przykładach. Portal *bit.ly* może wyświetlić takie ostrzeżenie: „Wydaje się, że ta strona może być szkodliwa, więc lepiej uważaj”.

HyperLogLog

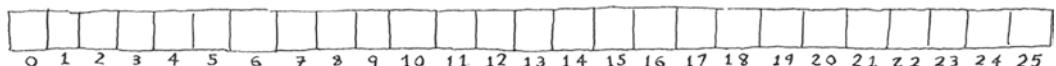
Podobne właściwości ma też algorytm o nazwie HyperLogLog. Wyobraź sobie, że Google chce policzyć *niepowtarzalne* wyszukiwania wygenerowane przez użytkowników. Albo powiedzmy, że Amazon chce sprawdzić, ile różnych przedmiotów obejrzał klienci sklepu. Szukanie odpowie na te pytania wymaga dużej ilości miejsca! W przypadku Google trzeba by zapisywać wszystkie niepowtarzające się wyszukiwane frazy. Gdy użytkownik czegoś szuka, trzeba sprawdzić, czy to coś znajduje się już w bazie danych. Jeśli nie, trzeba to dodać. Już w ciągu jednego dnia taka baza urosłaby do bardzo dużych rozmiarów!

HyperLogLog podaje przybliżoną liczbę niepowtarzających się elementów w zbiorze. Podobnie jak filtry Bloom'a, nie zwraca dokładnego wyniku, ale bardzo bliski prawdy, i zajmuje tylko ułamek pamięci potrzebnej do wykonania tego zadania w inny sposób.

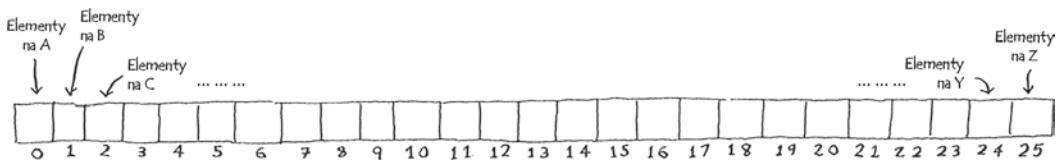
Jeśli masz bardzo dużo danych i zadowala Cię przybliżony wynik, koniecznie wypróbuj algorytmy probabilistyczne!

Algorytmy SHA

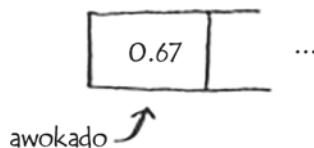
Pamiętasz algorytm obliczania skrótów z rozdziału 5.? Dla przypomnienia, wyobraź sobie, że masz klucz i chcesz wstawić do tablicy wartość.



Z pomocą funkcji obliczania skrótów wybierasz miejsce do wstawienia tej wartości.



W końcu wstawiasz posiadaną wartość do odpowiedniej komórki.



W ten sposób zapewniasz sobie stały czas wyszukiwania elementów. Kiedy chcesz sprawdzić wartość klucza, ponownie możesz użyć funkcji obliczania skrótów, która w czasie O(1) zwróci informację, jaką komórkę należy sprawdzić.

W takim przypadku staramy się, aby nasza funkcja obliczania skrótów zapewniała równomierne rozmieszczenie kluczy. W związku z tym nasza funkcja pobiera łańcuch i zwraca numer komórki dla tego łańcucha.

Porównywanie plików

Funkcją obliczania skrótów jest też funkcja implementująca algorytm SHA (ang. *secure hash algorithm*). Dla podanego łańcucha funkcja ta zwraca jego skrót.

"cześć" \Rightarrow e2f05796...

Nazewnictwo może być w tym przypadku nieco mylące. SHA jest **funkcją obliczania skrótów**. Generuje skrót, który jest po prostu krótkim łańcuchem. Funkcja obliczania skrótów dla tablic skrótów pozwala powiązać łańcuchy z indeksami tablicowymi, natomiast funkcja SHA kojarzy łańcuchy z łańcuchami.

SHA dla każdego łańcucha generuje inny skrót.

"cześć" \Rightarrow e2f05796...

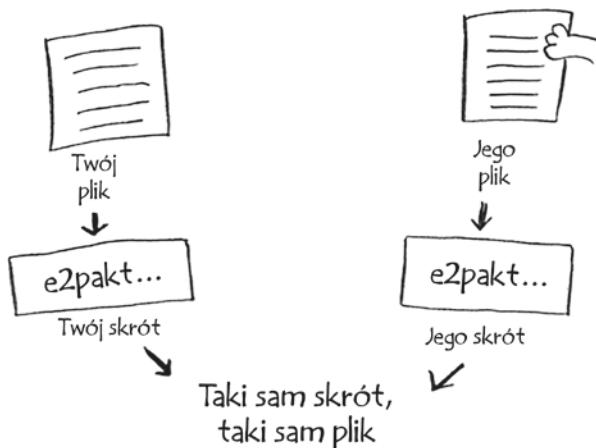
"algorytm" \Rightarrow cf13eb13...

"hasło" \Rightarrow 599459fd...

Uwaga

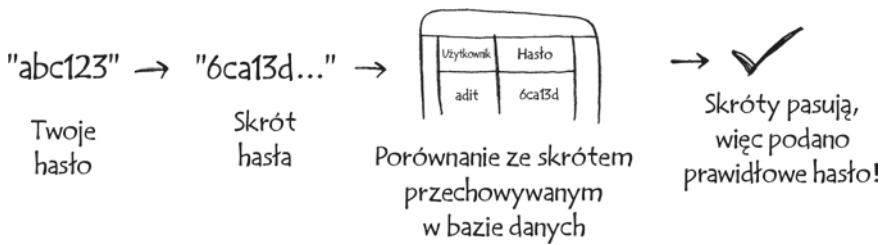
Skróty SHA naprawdę są długie. W podanych przykładach pokazano tylko fragmenty.

SHA pomaga stwierdzić, czy dwa pliki są takie same. Jest to wygodne rozwiązanie dla bardzo dużych plików. Powiedzmy, że mamy plik o rozmiarze 4 GB i chcemy się przekonać, czy nasz znajomy ma dokładnie taki sam plik. Nie musimy w tym celu wysyłać mu naszego pliku e-mailem, ponieważ wystarczy, że obie strony wygenerują skrót i porównają otrzymane wyniki.



Sprawdzanie haseł

Algorytm SHA można też wykorzystać do porównywaniałańcuchów bez ujawniania ich prawdziwej treści. Powiedzmy np., że hakerzy włamali się na serwery usługi Gmail i wykradli wszystkie hasła! Czy Twoje hasło jest bezbronne? Nie, nie jest. Google nie przechowuje oryginalnych haseł, tylko ich skróty SHA! Kiedy ktoś wpisze swoje hasło, Google oblicza jego skrót i porównuje go ze skrótem przechowywanym w swojej bazie danych.



Zatem skoro porównywane są tylko skróty, nie ma potrzeby przechowywać prawdziwych haseł! Algorytmu SHA bardzo często używa się do tego celu. Obliczanie skrótu hasła to proces jednokierunkowy, tzn. można obliczyć skrót hasła.

$\text{abc123} \rightarrow 6ca13d$

Jednak nie można obliczyć oryginalnego łańcucha na podstawie skrótu.

? → 6ca13d

Dzięki temu, jeśli nawet haker wykradnie skróty SHA z usługi Gmail, nie uda mu się ich zamienić na oryginalne hasła! Hasło można przekonwertować na skrót, ale odwrotny proces jest niemożliwy.

SHA to cała rodzina algorytmów: SHA-0, SHA-1, SHA-2 i SHA-3. Na razie wiadomo, że SHA-0 i SHA-1 mają pewne słabe punkty. Do obliczania skrótów haseł należy zatem używać SHA-2 lub SHA-3, choć aktualnie złotym standardem w dziedzinie obliczania skrótów haseł jest algorytm bcrypt (aczkolwiek nawet najlepszy algorytm nic nie da, jeśli użytkownik nie będzie rozważny).

Locality-sensitive hashing

Algorytm SHA ma jeszcze jedną bardzo ważną cechę: jest lokalnie nieczuły. Powiedzmy, że mamy łańcuch i generujemy dla niego skrót.

kot → b6a5ff

Jeśli w łańcuchu zmienimy choćby jedną literę, to skrót dla nowego łańcucha będzie całkowicie inny!

kop → 8166be

To bardzo dobra cecha, ponieważ dzięki niej haker nie odgadnie hasła, porównując jego skrót z innym, podobnym skrótem.

Czasami jednak pożądana jest odwrotna właściwość, tzn. funkcje czułe lokalnie też bywają przydatne. Jeśli potrzebujesz takiej funkcji, z pomocą przychodzi **Simhash**. Jeśli łańcuch zmieni się tylko nieznacznie, jego skrót również będzie się różnić od poprzedniego tylko w niewielkim stopniu. To umożliwia porównywanie skrótów, aby dowiedzieć się np., jak bardzo różnią się między sobą dwa łańcuchy, co jest bardzo przydatne!

- Google używa algorytmu Simhash do wykrywania duplikatów podczas przeglądania internetu.

- Nauczyciel przy użyciu algorytmu Simhash mógłby sprawdzić, czy uczeń nie skopiował pracy pisemnej z internetu.
- Użytkownicy portalu Scribd mogą wysyłać na serwery dokumenty i książki, którymi chcą się podzielić z innymi. Jednak właściciele portalu nie akceptują tekstu chronionych prawami autorskimi! Dlatego zaimplementowali algorytm Simhash, aby sprawdzać, czy np. treść wysyłanej książki nie jest taka sama jak Harry'ego Pottera i jeśli podejrzenia się potwierdzą, automatycznie ją odrzucają.

Algorytm Simhash umożliwia sprawdzanie podobieństwa elementów.

Wymiana kluczy Diffiego-Hellmana

Algorytm Diffiego-Hellmana zasługuje na to, by o nim wspomnieć, ponieważ w bardzo elegancki sposób rozwiązuje stary problem. Jak zaszyfrować wiadomość, aby mogła ją odczytać tylko ta osoba, dla której jest przeznaczona?

Najprościej wymyślić jakiś szyfr w rodzaju $a = 1, b = 2$ itd. Jeśli wyśle wiadomość o treści „1,12,1”, to będzie można rozszyfrować, że chodzi o napis „ala”. Aby jednak ta metoda była skuteczna, strony muszą uzgodnić między sobą sposób szyfrowania. Nie można tego zrobić przez e-mail, ponieważ ktoś może się włamać na naszą skrzynkę, odkryć szyfr i rozszyfrować wszystkie nasze wiadomości. Co gorsza, nawet osobiste spotkanie niewiele da, ponieważ ten szyfr jest bardzo prosty. Dlatego codziennie musielibyśmy zmieniać szyfr na inny, ale wówczas każdego dnia musielibyśmy się osobiście spotykać!

Gdyby nawet udawało się codziennie zmieniać szyfr, hakerzy nie mieliby problemu z jego złamaniem najprostszą metodą brutalnej siły. Powiedzmy, że przechwytyuję wiadomość o następującej treści: „22,10,20,2,11”. Najpierw próbuję szyfru $a = 1, b = 2$ itd.

$$\begin{array}{ccccc} 2 & 2 & 1 & 0 & 2 & 0 & 2 & 1 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ z & j & u & b & k \end{array}$$

To bez sensu, więc sprawdzam szyfr $a = 2, b = 3$ itd.

22	10	20	2	11
↓	↓	↓	↓	↓
w	i	t	a	j

Udało się! Taki prosty szyfr można bardzo łatwo złamać. Niemcy podczas II wojny światowej posługiwali się znacznie bardziej skomplikowanym szyfrem, a i tak został złamany. Wszystkie omówione problemy rozwiązuje algorytm Diffiego-Hellmana.

- Żadna ze stron nie zna szyfru, więc nie trzeba się spotykać, aby cokolwiek uzgodnić.
- Zaszyfrowana wiadomość jest *ekstremalnie* trudna do rozszyfrowania.

W algorytmie Diffiego-Hellmana wykorzystywane są dwa klucze — publiczny i prywatny. Klucz publiczny to po prostu taki klucz, który jest ogólnodostępny. Można go opublikować na stronie internetowej, wysłać komuś e-mailem albo zrobić z nim cokolwiek innego. Nie trzeba go ukrywać. Gdy ktoś chce wysłać nam wiadomość, musi ją zaszyfrować przy użyciu klucza publicznego. Zaszyfrowaną tym kluczem wiadomość można jednak rozszyfrować tylko przy użyciu klucza prywatnego. Dopóki tylko my znamy klucz prywatny, dopóty tylko my mamy możliwość rozszyfrowania przekazu!

Algorytm Diffiego-Hellmana jest wykorzystywany w praktyce wraz ze swoim następcą — algorytmem RSA. Jeśli interesujesz się kryptografią, warto na początek zapoznać się właśnie z algorytmem Diffiego-Hellmana — jest elegancki i niezbyt trudny do zrozumienia.

Programowanie liniowe

Najlepsze zostawiłem na koniec. Programowanie liniowe to jedna z najfajniejszych technologii, jakie znam.

Za pomocą programowania liniowego można coś zmaksymalizować na podstawie pewnych kryteriów. Powiedzmy np., że prowadzimy firmę wytwarzającą dwa produkty — koszule i płócienne torby. Do wyprodukowania koszuli potrzeba metra materiału i pięciu guzików, a do wyprodukowania torby — dwóch metrów materiału i dwóch guzików. Mamy 11 metrów materiału i 20

guzików. Na każdej koszuli zarabiamy 2 zł, a na torbie — 3 zł. Ile koszul i ile toreb powinniśmy wyprodukować, aby zarobić jak najwięcej?

W tym przypadku chcemy zmaksymalizować zysk, a ogranicza nas ilość posiadanych materiałów.

Inny przykład: jesteś politykiem i chcesz zdobyć jak największą liczbę głosów. Z Twoich badań wynika, że zdobycie jednego głosu w Warszawie wymaga godziny pracy (marketing, badania itd.), a we Wrocławiu — półtorej godziny. Potrzebujesz przynajmniej 500 głosów z Warszawy i przynajmniej 300 z Wrocławia. Masz 50 dni. Ponadto zdobycie głosu w Warszawie kosztuje 2 zł, a we Wrocławiu — 1 zł. Wysokość Twojego budżetu to 1500 zł. Ile maksymalnie możesz zdobyć głosów (łącznie w Warszawie i Wrocławiu)?

W tym przypadku chcemy osiągnąć maksymalną liczbę głosów, a ograniczają nas czas i pieniądze.

Pewnie sobie myślisz: „W tej książce temat optymalizacji był poruszany już wiele razy. Ciekawe, jak te wszystkie problemy mają się do programowania liniowego?”. Wszystkie algorytmy grafowe można zrealizować także za pomocą programowania liniowego. Technika ta jest bardziej ogólnym modelem, a problemy grafowe są jego częściami. Pewnie masz już dość!

W programowaniu liniowym wykorzystuje się algorytm o nazwie *simpleks*. Jest skomplikowany i dlatego w książce nie zamieściłem jego opisu. Jeśli interesujesz się optymalizacją, koniecznie poczytaj więcej o programowaniu liniowym!

Epilog

Mam nadzieję, że ten zwięzły przegląd 10 algorytmów uświadomił Ci tylko, jak wiele jest jeszcze do odkrycia. Uważam, że najlepszą metodą nauki jest znalezienie czegoś interesującego i zgłębienie tajników. Czytając tę książkę, właśnie zdobyłeś solidne podstawy do dalszej nauki na własną rękę.





ROZDZIAŁ 1.

- 1.1.** Powiedzmy, że mamy posortowaną listę 128 nazwisk i chcemy ją przeszukać za pomocą algorytmu wyszukiwania binarnego. Ile maksymalnie prób zgadywania będzie trzeba wykonać?

Rozwiązanie: 7

- 1.2.** Ile maksymalnie prób zgadywania trzeba będzie wykonać, jeśli podwoi się liczbę elementów w liście?

Rozwiązanie: 8

- 1.3.** Dane jest nazwisko i trzeba znaleźć numer telefonu osoby o tym nazwisku w książce telefonicznej.

Rozwiązanie: $O(\log n)$.

- 1.4.** Dany jest numer telefonu i trzeba znaleźć nazwisko właściciela tego numeru w książce telefonicznej. (Podpowiedź: musisz przeszukać całą książkę!).

Rozwiązanie: $O(n)$

- 1.5.** Chcesz przeczytać numery wszystkich osób w książce telefonicznej.

Rozwiązanie: $O(n)$

- 1.6.** Chcesz przeczytać numery tylko osób o nazwiskach na A. (Tu jest pułapka! Trzeba posłużyć się wiedzą przedstawioną bardziej szczegółowo w rozdziale 4. Przeczytaj odpowiedź — może Cię zaskoczy!).

Rozwiązanie: $O(n)$. Może się wydawać, że wykonuję operację tylko dla jednego z 26 znaków, więc czas wykonywania powinien wynosić

$O(n/26)$. Jednak istnieje zasada, w myśl której wszelkie dodawania, odejmowania, mnożenia i dzielenia się ignoruje. Żaden z tych zapisów nie jest poprawny: $O(n + 26)$, $O(n - 26)$, $O(n \cdot 26)$, $O(n / 26)$. Wszystkie są równoznaczne z $O(n)$! Jeśli ciekawi Cię, dlaczego tak jest, zajrzyj do podrozdziału „Jeszcze raz o notacji dużego O” w rozdziale 4. i poczytaj o stałych w notacji dużego O (stała to po prostu liczba, jak np. 26 w tym zadaniu).

ROZDZIAŁ 2.

- 2.1.** Wyobraź sobie, że tworzysz aplikację do zarządzania domowymi finansami.

1. warzywa
2. film
3. członkostwo w sfbc

Codziennie zapisujesz wszystkie swoje wydatki. Na koniec miesiąca przeglądasz notatki i podsumowujesz, ile wydałeś. W związku z tym często dodajesz nowe elementy i rzadko odczytyujesz dane. Czy w takim razie dla Ciebie lepsza będzie tablica, czy lista?

Rozwiązanie: W tym przypadku codziennie dodajemy do listy wydatki i raz w miesiącu odczytujemy wszystkie wydatki. Tablice charakteryzują się szybkim odczytem i powolnym wstawianiem. Listy powiązane mają powolne operacje odczytu i szybkie operacje wstawiania. Ponieważ częściej będziemy wstawiać elementy niż je odczytywać, lepszym wyborem będą listy powiązane. Ponadto operacja odczytu w listach powiązanych jest powolna tylko przy pobieraniu losowych elementów. My będziemy odczytywać *wszystkie* pozycje, więc nasza lista będzie szybka także w czasie *odczytu*. Zatem z pewnością w tej sytuacji lista powiązana jest dobrym wyborem.

- 2.2.** Wyobraź sobie, że tworzysz aplikację do przyjmowania zamówień od klientów w restauracji. Jedeną z funkcji jest zapisywanie listy zamówień. Kelnerzy cały czas dodają nowe zamówienia, a szefowie kuchni je pobierają i przygotowują potrawy. Jest to kolejka zamówień: kelnerzy dodają zamówienia na końcu kolejki, a szefowie kuchni pobierają je z początku kolejki i gotują dania.



Czy do implementacji takiej kolejki użyłbyś tablicy, czy listy powiązanej. (Podpowiedź: listy powiązane są dobre we wstawianiu i usuwaniu elementów, a tablice są lepsze w dostępie swobodnym. Które operacje będziesz wykonywać w tym przypadku?).

Rozwiązanie: Należy użyć listy powiązanej. Do struktury często trzeba wstawiać elementy (kelnerzy dodają zamówienia), a w tym listy powiązane są najlepsze. Nie trzeba niczego wyszukiwać ani mieć swobodnego dostępu do elementów (w czym najlepsze są tablice), ponieważ szefowie kuchni zawsze pobierają zamówienia po kolei.

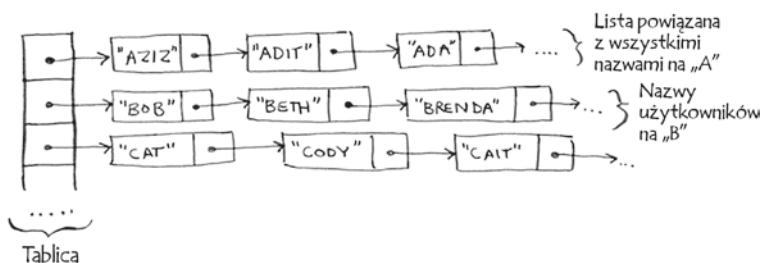
- 2.3.** Proponuję eksperiment myślowy. Powiedzmy, że Facebook przechowuje listę nazw użytkowników. Gdy ktoś próbuje zalogować się na konto, system szuka podanej przez tego kogoś nazwy użytkownika. Jeśli ją znajdzie, może nastąpić zalogowanie. Ludzie bardzo często logują się na Facebook, więc lista nazw użytkowników jest bardzo często przeszukiwana. Powiedzmy, że do jej przeszukiwania portal używa algorytmu wyszukiwania binarnego. Algorytm ten wymaga dostępu swobodnego — musi mieć natychmiastowy dostęp do środkowego elementu listy. Czy dysponując tą wiedzą, zaimplementowałbyś listę nazw użytkowników na bazie tablicy, czy listy powiązanej?

Rozwiązanie: Tablica posortowana. Tablice zapewniają swobodny dostęp do elementów, tzn. w każdej chwili można błyskawicznie uzyskać dostęp do dowolnego elementu tablicy. W przypadku list powiązanych nie jest to możliwe. Aby dostać się do elementu w środku takiej listy, należy najpierw przejść do pierwszego, a następnie podążać szlakiem połączeń aż do szukanej pozycji.

- 2.4.** Na Facebooku bardzo często tworzone są nowe konta. Powiedzmy, że listę użytkowników postanowiliśmy przechowywać w tablicy. Jakie są wady tablicy, jeśli chodzi o wstawianie elementów? Innymi słowy, pomyśl sobie, że szukasz loginów za pomocą wyszukiwania binarnego. Co się stanie, gdy dodasz nowych użytkowników do tablicy?

Rozwiązańe: Wstawianie elementów do tablic jest powolne. Ponadto, jeśli użytkownicy są wyszukiwani za pomocą wyszukiwania binarnego, tablica musi być posortowana. Wyobraź sobie, że na Facebooku rejestruje się ktoś podający nazwisko Adit B. To nazwisko zostanie wstawione na końcu tablicy, więc trzeba ją posortować za każdym razem, gdy ktoś doda element.

- 2.5.** W rzeczywistości Facebook nie przechowuje informacji ani w tablicy, ani w liście powiązanej. Zastanówmy się więc nad hybrydową strukturą danych: tablicą list powiązanych. Mamy tablicę z 26 miejscami. Każde z nich wskazuje listę powiązaną. Pierwsze miejsce np. wskazuje listę powiązaną zawierającą wszystkie nazwy użytkowników zaczynające się od litery a. Drugie miejsce wskazuje listę powiązaną zawierającą wszystkie nazwy użytkowników zaczynające się od litery b itd.



Powiedzmy, że na Facebooku postanawia założyć konto Adit B i chce my dodać go do listy. Przechodzimy do 1. miejsca w tablicy, następnie idziemy do wskazywanej przez niego listy powiązanej i dodajemy Adit B na końcu. A teraz wyobraź sobie, że chcesz znaleźć użytkownika o nazwie Zakhir H. Idziemy więc do miejsca 26. w tablicy, które wskazuje listę powiązaną zawierającą wszystkie nazwy na Z. Następnie w liście tej szukamy użytkownika Zakhir H.

Porównaj tę strukturę danych z tablicami i listami powiązanymi. Czy będzie szybsza, czy wolniejsza, jeśli chodzi o wyszukiwanie i wstawianie danych w porównaniu z tymi strukturami? Nie musisz podawać czasów wykonywania w notacji dużego O — wystarczy, że powiesz, czy ta nowa struktura będzie szybsza, czy wolniejsza.

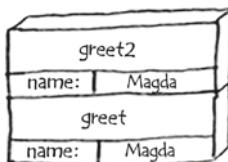
Rozwiązańe: Wyszukiwanie — wolniejsze niż w tablicach, szybsze niż w listach powiązanych. Wstawianie — szybsze niż w tablicach, tak samo szybkie jak w listach powiązanych. Zatem nasza struktura jest wolniejsza od tablic, jeśli chodzi o wyszukiwanie, ale szybsza od nich i tak samo szybka jak listy powiązane we wszystkich operacjach. Dalej

w książce opisana jest jeszcze inna hybrydowa struktura danych zwana tablicą skrótów. Ten przykład powinien uświadomić Ci, jak tworzy się złożone struktury danych na podstawie prostszych.

Czego w takim razie używa Facebook? Najprawdopodobniej ma wiele baz danych opartych na różnych strukturach danych: tablicach skrótów, B-drzewach itd. Tablice i listy powiązane są cegiełkami, z których tworzy się bardziej złożone struktury danych.

ROZDZIAŁ 3.

3.1. Spójrz na poniższy stos wywołań.



Jakie informacje możesz podać na podstawie tego stosu?

Rozwiązanie: Oto kilka przykładowych wniosków.

- Na początku została wywołana funkcja `greet` z parametrem `name = magda`.
- Następnie funkcja `greet` wywołała funkcję `greet2` z parametrem `name = magda`.
- W tym momencie wykonywanie funkcji `greet` jeszcze się nie zakończyło i znajduje się ona w stanie zawieszenia.
- Aktualnie wykonywane jest wywołanie funkcji `greet2`.
- Po zakończeniu wykonywania tej funkcji następuje wznowienie wykonywania funkcji `greet`.

3.2. Wyobraź sobie, że przez przypadek napisałeś nieskończoną funkcję rekurencyjną. Jak już wiesz, dla każdego wywołania funkcji komputer przydziela pamięć na stosie. Co się dzieje ze stosem, gdy funkcja nigdy się nie kończy?

Rozwiązanie: Stos nigdy nie przestanie rosnąć. Każdy program ma przydzieloną ograniczoną ilość miejsca na stosie wywołań. Kiedy program wyczerpie całą dostępną mu pamięć (w końcu do tego dojdzie), ulegnie awarii z powodu błędu przepełnienia stosu.

ROZDZIAŁ 4.

- 4.1.** Napisz kod źródłowy wcześniejszej funkcji `sum`.

Rozwiżanie:

```
def sum(list):
    if list == []:
        return 0
    return list[0] + sum(list[1:])
```

- 4.2.** Napisz funkcję rekurencyjną liczącą elementy w liście.

Rozwiżanie:

```
def count(list):
    if list == []:
        return 0
    return 1 + count(list[1:])
```

- 4.3.** Znajdź największą liczbę w liście.

Rozwiżanie:

```
def max(list):
    if len(list) == 2:
        return list[0] if list[0] > list[1] else list[1]
    sub_max = max(list[1:])
    return list[0] if list[0] > sub_max else sub_max
```

- 4.4.** Pamiętasz wyszukiwanie binarne z rozdziału 1.? To też jest algorytm typu „dziel i rządź”. Potrafisz określić przypadki bazowy i rekurencyjny dla wyszukiwania binarnego?

Rozwiżanie: Przypadek bazowy wyszukiwania binarnego to tablica zawierająca jeden element. Jeżeli szukany element jest taki sam jak ten, który znajduje się w tablicy, to go znajdziemy. W przeciwnym przypadku szukanego elementu nie ma strukturze.

W rekurencyjnym przypadku wyszukiwania binarnego tablicę dzieli się na pół, jedną połowę się odrzuca i wywołuje się wyszukiwanie binarne na drugiej.

Przedstaw czas wykonywania wszystkich poniższych operacji w notacji dużego O.

- 4.5.** Drukowanie wartości wszystkich elementów w tablicy.

Rozwiżanie: $O(n)$

- 4.6.** Podwojenie wartości każdego elementu w tablicy.

Rozwiązanie: $O(n)$

- 4.7.** Podwojenie wartości tylko pierwszego elementu tablicy.

Rozwiązanie: $O(1)$

- 4.8.** Utworzenie tabliczki mnożenia dla wszystkich elementów w tablicy.

Jeśli np. dana jest tablica [2, 3, 7, 8, 10], najpierw mnożysz wszystkie elementy przez 2, potem przez 3, potem przez 7 itd.

Rozwiązanie: $O(n^2)$

ROZDZIAŁ 5.

Które z poniższych funkcji są konsekwentne?

- 5.1.** $f(x) = 1$ $\leftarrow \dots$ Zwraca „1” dla wszystkich danych wejściowych.

Rozwiązanie: konsekwentna

- 5.2.** $f(x) = \text{rand}()$ $\leftarrow \dots$ Za każdym razem zwraca losową liczbę.

Rozwiązanie: niekonsekwentna

- 5.3.** $f(x) = \text{next_empty_slot}()$ $\leftarrow \dots$ Zwraca indeks następnej pustej komórki w tablicy skrótów.

Rozwiązanie: niekonsekwentna

- 5.4.** $f(x) = \text{len}(x)$ $\leftarrow \dots$ Jako indeksu używa długości łańcucha.

Rozwiązanie: konsekwentna

Wyobraź sobie, że masz cztery funkcje obliczania skrótów pobierające łańcuchy. Oto one.

- Funkcja zwracająca 1 dla wszystkich danych wejściowych.
- Funkcja wykorzystująca jako indeks długość otrzymanego na wejściu łańcucha.
- Funkcja wykorzystująca jako indeks pierwszą literę otrzymanego na wejściu łańcucha, tak że wszystkie napisy zaczynające się na *a* są grupowane w jednej komórce itd.
- Funkcja zamieniająca każdą literę na liczbę pierwszą: $a = 2, b = 3, c = 5, d = 7, e = 11$ itd. Dla podanego łańcucha funkcja obliczania skrótów oblicza sumę wszystkich znaków i dzieli ją bez reszty przez rozmiar tablicy skrótów. Jeśli np. rozmiar tablicy skrótów wynosi 10 i zostanie przekazany łańcuch „gad”, indeks tego łańcucha wyniesie $17 + 2 + 7 \% 10 = 26 \% 10 = 2$.

Która z powyższych funkcji zapewniłaby dobry rozkład w poniższych przykadkach przy założeniu, że rozmiar tablicy skrótów wynosi 10 miejsc?

- 5.5.** Książka telefoniczna, w której kluczami są imiona, a wartościami numery telefonów. Dane są następujące imiona: Eliza, Bernard, Bożydar i Daniel.

Rozwiązańe: Dobry rozkład zapewniłyby funkcje C i D.

- 5.6.** Przypisanie rozmiaru baterii do jej mocy. Dostępne są rozmiary A, AA, AAA oraz AAAA.

Rozwiązańe: Dobry rozkład zapewniłyby funkcje B i D.

- 5.7.** Przypisanie tytułów książek do autorów. Dane są następujące tytuły: *Maus*, *Fun Home* i *Watchmen*.

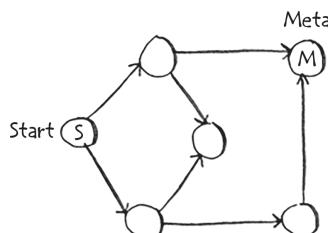
Rozwiązańe: Dobry rozkład zapewniłyby funkcje B, C i D.

ROZDZIAŁ 6.

Przeprowadź wyszukiwanie wszerz każdego z poniższych grafów, aby znaleźć rozwiązanie.

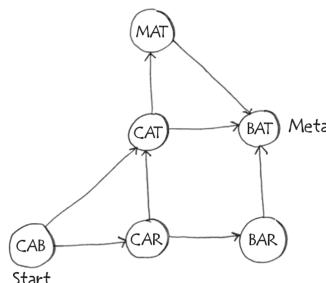
- 6.1.** Znajdź długość najkrótszej drogi od linii startu do mety.

Rozwiązańe: długość najkrótszej drogi wynosi 2.

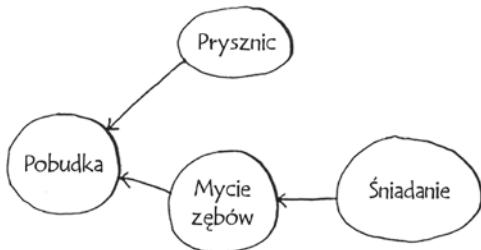


- 6.2.** Znajdź długość najkrótszej drogi z „cab” do „bat”.

Rozwiązańe: długość najkrótszej drogi wynosi 2.



6.3. Oto niewielki graf przedstawiający mój typowy poranek.



Przyjrzyj się trzem poniższym listom i powiedz, które z nich są poprawne.

A.

1. Pobudka
2. Prysznic
3. Śniadanie
4. Mycie zębów

B.

1. Pobudka
2. Mycie zębów
3. Śniadanie
4. Prysznic

C.

1. Prysznic
2. Pobudka
3. Mycie zębów
4. Śniadanie

Rozwiązanie: A — niepoprawna; B — poprawna, C — niepoprawna

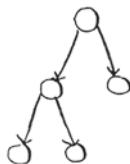
6.4. Oto nieco większy graf. Utwórz z niego poprawną listę.



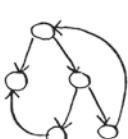
Rozwiązanie: 1 — pobudka; 2 — ćwiczenia; 3 — prysznic; 4 — mycie zębów; 5 — ubieranie się; 6 — spakowanie drugiego śniadania; 7 — śniadanie

6.5. Które z poniższych grafów są także drzewami?

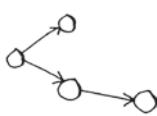
A.



B.



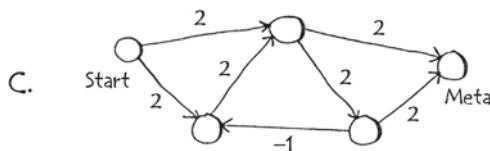
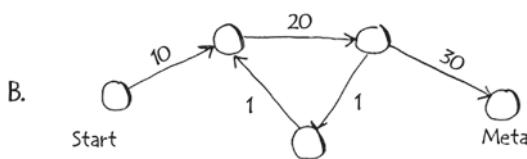
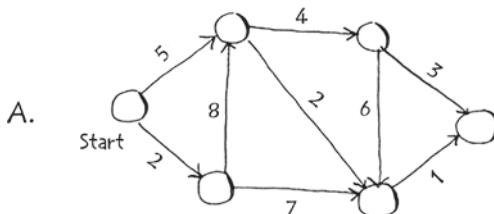
C.



Rozwiązanie: A — drzewo; B — niedrzewo; C — drzewo. Ostatni graf przedstawia drzewo pochylone. Drzewa są podzbiorem grafów, więc każde drzewo jest grafem, ale nie każdy graf jest drzewem.

ROZDZIAŁ 7.

7.1. Określ wagę najkrótszej drogi od linii startu do mety na każdym z poniższym grafów.



Rozwiązanie: A: A — 8; B — 60; C — trudne pytanie. Nie da się obliczyć najkrótszej drogi z powodu ujemnej wagi.

ROZDZIAŁ 8.

- 8.1.** Pracujesz w firmie meblarskiej, w której rozwozisz produkty do różnych miejsc w kraju. Musisz załadować swoją ciężarówkę pudłami. Pudła te są różnych rozmiarów, więc za każdym razem zastanawiasz się, jak najlepiej wykorzystać dostępną w ciężarówce przestrzeń. Jak ładować pudła, aby wykorzystać miejsce do maksimum? Opracuj zachłanną strategię. Czy pozwala ona uzyskać optymalne rozwiązanie?

Rozwiązanie: Zachłanna strategia polegałaby na wybraniu największego pudła, które zmieści się w dostępnej przestrzeni i powtarzaniu tej czynności, aż nie będzie możliwości zapakowania żadnego następnego pudła. Nie, w ten sposób nie obliczymy optymalnego rozwiązania.

- 8.2.** Wybierasz się na wycieczkę po Europie i masz siedem dni na obejrzenie jak największej liczby miejsc. Każdemu miejscu przypisujesz wartość punktową (oznaczającą, jak bardzo chciałbyś je odwiedzić) oraz szacujesz, ile czasu zajmie dojazd. W jaki sposób uzyskać jak największą sumę punktów (aby dotrzeć do wszystkich miejsc, które chce się zobaczyć) podczas wycieczki? Opracuj zachłanną strategię. Czy pozwala ona uzyskać optymalne rozwiązanie?

Rozwiązanie: Wybieraj najwyżej punktowaną czynność, jaką możesz wykonać w czasie, który Ci pozostał. Zakończ pracę, gdy nie będzie już możliwości zrobienia czegokolwiek więcej. Nie, w ten sposób nie obliczymy optymalnego rozwiązania.

Dla każdego z tych algorytmów wskaż, czy jest on zachłanny, czy nie.

- 8.3.** Szybkie sortowanie

Rozwiązanie: nie

- 8.4.** Wyszukiwanie wszerz

Rozwiązanie: tak

- 8.5.** Algorytm Dijkstry

Rozwiązanie: tak

- 8.6.** Listonosz musi dostarczyć listy do 20 domów i chciałby się dowiedzieć, jaka jest najkrótsza trasa między tymi dwudziestoma miejscami. Czy jest to problem NP-zupełny?

Rozwiązanie: tak

- 8.7.** Wyszukiwanie największej kliki w zbiorze ludzi (tu przyjmujemy, że *klika* to zbiór ludzi, którzy wzajemnie się znają) — czy to jest problem NP-zupełny?

Rozwiązańe: tak

- 8.8.** Robisz mapę USA i sąsiadującym stanom chcesz nadać różne kolory. Musisz znaleźć najmniejszą liczbę kolorów, jaka wystarczy do pokolorowania stanów tak, żeby żadna para sąsiadów nie miała tego samego koloru. Czy to jest problem NP-zupełny?

Rozwiązańe: tak

ROZDZIAŁ 9.

- 9.1.** Powiedzmy, że możemy ukraść jeszcze jeden produkt — odtwarzacz MP3. Urządzenie waży 1 kg i kosztuje 1000 zł. Czy warto je brać?

Rozwiązańe: tak. Następnie można ukraść odtwarzacz MP3, iPhone i gitarę na łączną kwotę 4500 zł.

- 9.2.** Jedziesz na biwak. Masz plecak o obciążeniu 6 kg i możesz zabrać przedmioty z poniższej listy. Każdy z nich ma określoną wartość. Im ona wyższa, tym ważniejszy jest dany produkt.

- Woda, 3 kg, 10
- Książka, 1 kg, 3
- Jedzenie, 2 kg, 3
- Kurtka, 2 kg, 5
- Aparat, 1 kg, 6

Jaki jest optymalny zestaw przedmiotów do zabrania na wycieczkę?

Rozwiązańe: Należy wziąć wodę, jedzenie i aparat.

- 9.3.** Narysuj i wypełnij danymi siatkę do obliczania najdłuższego wspólnego łańcucha dla napisów *blue* i *clues*.

Rozwiązańe:

	C	L	U	E	S
B	o	o	o	o	o
L	o	l	o	o	o
U	o	o	2	o	o
E	o	o	o	3	o

ROZDZIAŁ 10.

- 10.1.** W przykładzie dotyczącym Netfliksa dystans dzielący dwóch użytkowników obliczaliśmy za pomocą specjalnego wzoru. Jednak nie wszyscy użytkownicy oceniają filmy w taki sam sposób. Powiedzmy, że mamy dwóch użytkowników, nazwijmy ich Yogi i Pinky, którzy mają taki sam gust filmowy. Mimo to Yogi wszystkim filmom, które mu się podobają, przyznaje piątkę, a Pinky jest bardziej wybredny i najwyższą ocenę przyznaje tylko naprawdę — jego zdaniem — najlepszym filmom. Choć panowie mają podobne gusty, według naszego algorytmu obliczania dystansu nie są sąsiadami. Co zrobisz, by uwzględnić w obliczeniach także te odmienne strategie oceniania?

Rozwiązańe: Można zastosować tzw. *normalizację*. Oblicza się średnią ocen wydawanych przez każdą osobę i na podstawie otrzymanego wyniku stosuje się skalowanie jej ocen. Można np. zauważyć, że średnia ocen Pinkiego wynosi 3, podczas gdy średnia ocen Yogiego to 3,5. Zatem podbijamy nieco oceny Pinkiego, aż jego średnia dojdzie także do 3,5. Teraz można porównywać oceny w tej samej skali.

- 10.2.** Powiedzmy, że Netflix wyznacza grupę „liczących się użytkowników”. Zaliczają się do nich np. Quentin Tarantino oraz Wes Anderson i ich oceny liczą się znacznie bardziej niż zwykłych użytkowników. Jak zmodyfikowałbyś system rekommendacji, aby bardziej zdecydowanie uwzględniał opinie takich osób?

Rozwiązańe: Przy użyciu algorytmu KNN ocenom autorytetów można nadać większą wagę. Powiedzmy, że mamy trzech sąsiadów: Jana, Dawida i Wesa Andersona (autorytet). Panowie ocenili film *Golfiarze* odpowiednio na poziomie 3, 4 i 5. Zamiast po prostu obliczyć średnią tych ocen ($3 + 4 + 5 / 3 = 4$ gwiazdki), ocenie Wesa Andersona można przypisać większą wagę, np. $3 + 4 + 5 + 5 + 5 / 5 = 4,4$ gwiazdki.

- 10.3.** Netflix ma miliony użytkowników. Opisany w przykładzie system rekommendacji opierał się na analizie cech pięciu najbliższych sąsiadów. Czy myślisz, że to za dużo? A może za mało?

Rozwiązańe: Za mało. Jeśli uwzględnisz mniej sąsiadów, istnieje większe ryzyko, że otrzymasz skrzywione wyniki. Istnieje dobra zasada, że jeśli jest N użytkowników, należy uwzględnić \sqrt{N} sąsiadów.



Skorowidz

A

algorytm, 1, 3
aproksymacyjny, 147, 157, 163
bcrypt, 216
Bellmana-Forda, 130
czas wykonywania, 10, 11, 12, 13, 16
liniowy, 10
logarytmiczny, 10
 $O(1)$, 89, 90, 214
 $O(2^n)$, 147, 162
 $O(\log n)$, 13, 14, 15, 16, 17, 73, 74, 205
 $O(n \log n)$, 15, 16, 35, 66, 71, 208
 $O(n!)$, 15, 16, 19
 $O(n)$, 12, 14, 15, 16, 17, 73, 74, 90, 205
 $O(n^2)$, 15, 16, 34, 66, 147
Diffiego-Hellmana, 217, 218
Dijkstry, 115, 116, 117, 120, 121, 123
implementacja, 131
dość dobry, 145
efektywność, 10
Euklidesa, 54
Feynmana, 180
grafowy, 96
HyperLogLog,
 Patrz: HyperLogLog
k najbliższych sąsiadów, 187, 189, 195, 197
wybór cech, 198
lokalnie nieczuły, 216
MapReduce, Patrz: MapReduce probabilistyczny, 212, 213

programowania
dynamicznego, Patrz:
programowanie dynamiczne
rekurencyjny, 53
rozproszony, 209
równoległy, 208, 209
RSA, 218
SHA, 213, 215, 216
Simhash, 216
simpleks, 219
sortowania
bardzo wolny, 16, 17
przez scalanie, 66, 67
szybkiego, 15, 35, 60, 65, 66, 68, 208
topologicznego, 112
wolny, 15
średni czas działania, 15
wyszukiwanie binarne, Patrz:
 wyszukiwanie binarne
 zachłanny, 141, 144, 147
Apache Hadoop, 209
aproksymacja, 157

B

B-drzewo, 206
Bellmana-Forda algorytm, Patrz:
 algorytm Bellmana-Forda
BFS, Patrz: wyszukiwanie wszerz
binary search tree, Patrz: drzewo binarne poszukiwań
Bloom filtr, 212
breadth-first search, Patrz:
 wyszukiwanie wszerz

C

caching, Patrz: pamięć podręczna
 zapisywanie
cykl, 121, 122

czas
liniowy, 15
logarytmiczny, 15
stały, 89

dictionary, Patrz: słownik
Diffiego-Hellmana
algorytm, Patrz: algorytm Diffiego-Hellmana
Dijkstry algorytm, Patrz:
algorytm Dijkstry
divide and conquer, Patrz:
 metoda dziel i rządź
dowód indukcyjny, 65
drzewo, 113, 201
B-drzewo, 206
binarne poszukiwań, 203
zrównoważone, 206
czerwono-czarne, 206

E

element
dostęp
 sekwencyjny, 30
 swobodny, 30, 205
osiowy, 60, 68
wstawianie, 205

F

Feynmana algorytm, Patrz:
 algorytm Feynmana
FIFO, 104, 114
filtr Blooma, 212
format
 JPG, 208
 MP3, 207
Fouriera transformata,
 Patrz: transformata Fouriera

funkcja

- map, 209
- obliczania skrótów, 75, 76, 78, 88, 93, 213, 214
- reduce, 210
- rekurencyjna, 40, *Patrz też:*
 - rekurencja
 - na tablicy, 58
 - przypadek podstawowy, 40, 41
 - przypadek rekurencyjny, 40, 41
- SHA, 93
- silnia, *Patrz:* silnia

G

- graf, 96, 98, 114, 188, 191
- cykl, *Patrz:* cykl
- implementacja, 105, 107
- krawędź, *Patrz:* krawędź
- nieważony, 120
- skierowany, 106, 114
 - acykliczny, 122
- sortowanie topologiczne, 112
- ważony, 120
- węzeł, *Patrz:* węzeł

H

- hash function, *Patrz:* funkcja obliczania skrótów
- hash table, *Patrz:* tablica skrótów
- hasło, 215
- HyperLogLog, 213

I

- indeks odwrócony, 206, 207
- inverted index, *Patrz:* indeks odwrócony

J

- język programowania funkcyjny, 59
- Haskell, 59

K

- klasyfikator bayesowski naiwny, 200
- klucz, 79, 148
 - prywatny, 218
 - publiczny, 218
- k-nearest neighbors, *Patrz:*
 - algorytm k najbliższych sąsiadów
- KNN, *Patrz:* algorytm k najbliższych sąsiadów
- kolejka, 103
- kolizja, 86, 87, 92
- krawędź, 99, 113
 - waga, 120
 - ujemna, 128
- kryptografia, 218

L

- Levenshteina odległość, *Patrz:*
 - odległość Levenshteina
- LIFO, 104, 114
- lista, 23
 - czas wykonywania operacji, 28, 30
- element
 - dostęp, 30
 - usuwanie, 30
 - wstawianie, 29
- posortowana, 3, 8
- powiązana, 25, 88
 - wady, 27
- logarytm, 7

Ł

- łańcuch, 214
- najdłuższa wspólna część, 178, 184
- porównywanie, 215
- stopień podobieństwa, 185

M

- MapReduce, 209

merge sort, *Patrz:* sortowanie

przez scalanie

metoda dziel i rządź, 51, 52, 56, 60, 71

N

- naive Bayes classifier, *Patrz:*
 - klasyfikator bayesowski naiwny
 - najdłuższa wspólna część
 - łańcucha, 178
- najdłuższa wspólna
 - podsekwencja, 184
- notacja dużego O, 7, 10, 12, 15, 16, 17, 66
- stała, 35, 67, 68
- null, *Patrz:* wartość null

O

- OCR, *Patrz:* optyczne
 - rozpoznawanie znaków
 - odległość Levenshteina, 185
- operacja
 - pop, 42
 - push, 42
- optical character recognition,
 - Patrz:* optyczne rozpoznawanie znaków
 - optyczne rozpoznawanie znaków, 199, 200
 - szkolenie, 200
- optymalizacja, 219
 - planu podróży, 176

P

- pamięć
 - adres, 23
 - podręczna zapisywanie, 84
- partitioning, *Patrz:*
 - partyjonowanie
 - partyjonowanie, 61
- Pitagorasa twierdzenie, *Patrz:*
 - twierdzenie Pitagorasa
- pivot, *Patrz:* element osiowy
- podobieństwo kosinusowe, 197

problem
 komiwojażera, 16, 17, 153,
 154, 157
 NP-zupełny, 152, 153, 157,
 158, 159
 plecaka, 144, 161, 171, 174,
 175, 178
 podróżującego komiwojażera,
Patrz: problem
 komiwojażera
 pokrycia zbioru, 146, 157, 158
 wyboru najkrótszej drogi,
 programowanie
 dynamiczne, 161, 163, 176,
 177, 178, 179, 183, 185, 186
 funkcyjne, 59
 liniowe, 218
 przewidywanie, 201
 pseudokod, 38, 40

Q

quicksort, *Patrz:* algorytm
 sortowania szybkiego

R

regresja, 196
 rekomendacja, 189, 190, 194
 rekurencja, 37, 38, 39, 40, 45,
Patrz też: funkcja rekurencyjna
 ogonowa, 49
 rodzic, 124
 rozpoznawanie nazw DNS, 81

S

secure hash algorithm,
Patrz: algorytm SHA
 shortest-path problem,
Patrz: problem wyboru
 najkrótszej drogi
 silnia, 19, 156, 157
 słownik, 78
 sortowanie
 przez scalanie, *Patrz:* algorytm
 sortowania przez scalanie
 przez wybieranie, 15, 32

szynkowe, *Patrz:* algorytm
 sortowania szybkiego
 spam, 200
 stos, 42
 wywołań, 42, 43, 69, 70
 z rekurencją, 45
 struktura danych
 probabilistyczna, 212
 szyfr, 217

T

tablica, 8, 23, 24, 26, 210
 czas wykonywania operacji,
 28, 30
 element
 dostęp, 30
 usuwanie, 30
 wstawianie, 29
 indeks, 28
 mieszająca, *Patrz:* tablica
 skrótów
 skrótów, 78, 105, 207, 212
 jako pamięć podręczna, 83
 przeszukiwanie, 80
 współczynnik zapełnienia,
 wydajność, 88, 89
 zmiana rozmiaru, 91
 zalety, 27

tail recursion, *Patrz:* rekurencja
 ogonowa
 transformata Fouriera, 207
 traveling salesman problem,
Patrz: problem komiwojażera
 twierdzenie Pitagorasa, 191

U

uczenie maszynowe, 199, 200, 201

W

wartość null, 3
 węzeł, 99, 113
 koszt, 133
 najtańszy, 117, 120
 sąsiad, 99, 120
 współczynnik zapełnienia, 90

wyszukiwanie
 binarne, 3, 5, 8, 73, 203
 czas wykonywania, 11
 liczba prób, 6, 13, 15
 proste, 5, 73
 czas wykonywania, 11
 liczba prób, 6
 wszerz, 95, 98, 99, 102, 121
 czas wykonywania, 111
 wyszukiwarka internetowa, 206

Z

zapytanie SQL, 209
 zbiór, 149, 150, 151
 potęgowy, 146
 przecięcie, 150, 151
 różnica, 150
 suma, 150
 znak rozpoznawanie optyczne,
Patrz: optyczne rozpoznawanie
 znaków

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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 WYDAWNICZA

 **Helion SA**