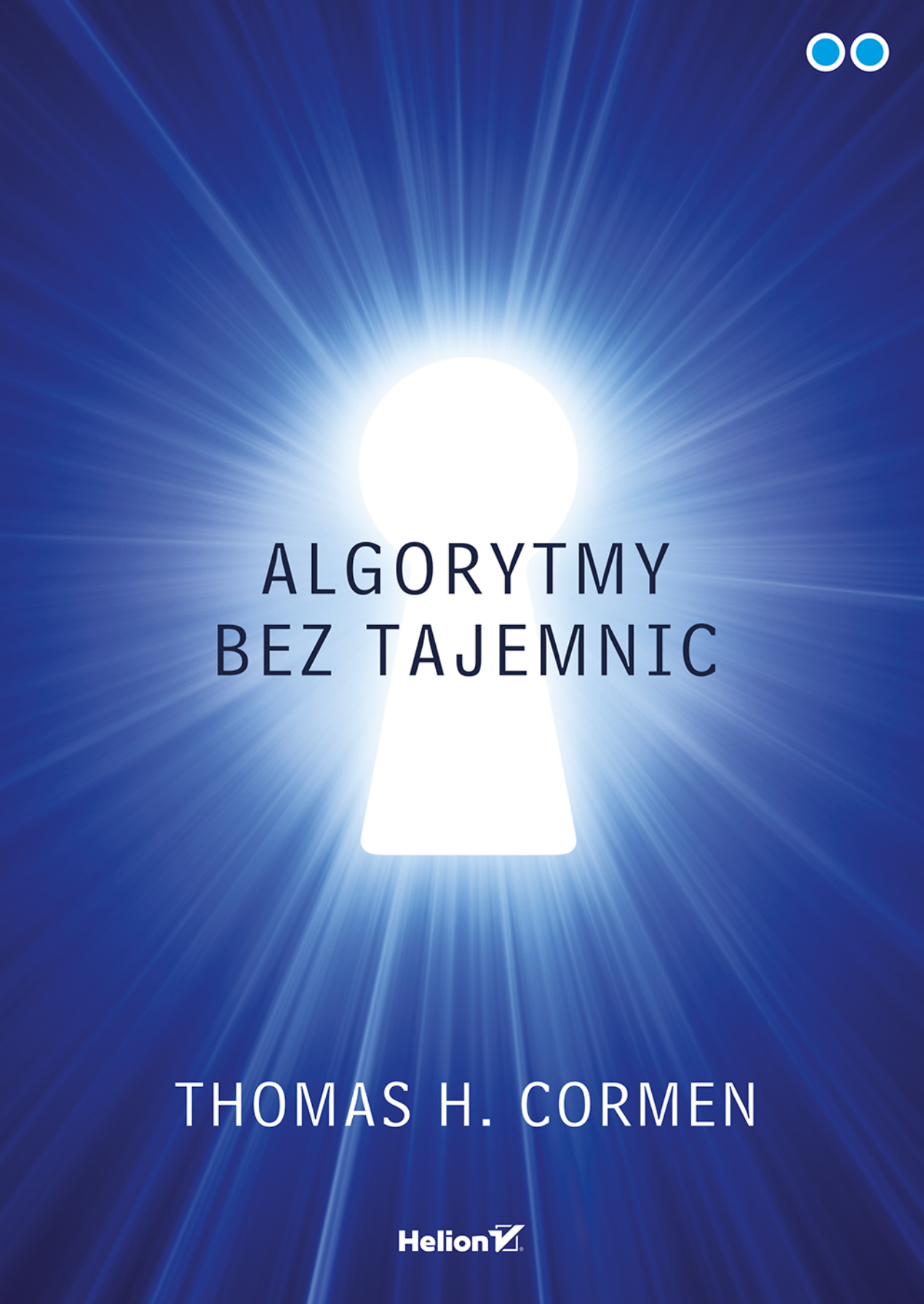
ALGORYTMY
BEZ TAJEMNIC
THOMAS H. CORMEN
Helion

Tytuł oryginału: Algorithms Unlocked

Tłumaczenie: Zdzisław Płoski

ISBN: 978-83-283-6736-4

Helion SA
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)

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

Printed in Poland.

Spis treści

1 Co to są algorytmy i dlaczego warto poświęcać im uwagę?

Zacznijmy od pytania, które mi często zadają: „Co to jest algorytm?”[1].

Ogólna odpowiedź mogłaby być taka: „zbiór kroków prowadzących do wykonania zadania”. Na co dzień stosujesz różne algorytmy. Masz algorytm mycia zębów: otwierasz tubkę z pastą, bierzesz szczoteczkę do ręki, wyciskasz na szczoteczkę tyle pasty, ile trzeba, zamykasz tubkę, wkładasz szczoteczkę do jednej ćwiartki paszczy, przesuwasz nią w górę i w dół (oraz w prawo i w lewo) przez *N* sekund itd. Jeśli musisz dojeżdżać do pracy, masz swój algorytm dojazdu do pracy. I tak dalej.

Jednak ta książka zajmuje się algorytmami, które działają na komputerach, a ogólnie rzecz ujmując — na urządzeniach obliczeniowych. Z algorytmami wykonywanymi na komputerach jest podobnie jak z tymi, które wykonujesz na co dzień. Korzystasz z GPS-u, żeby znaleźć drogę? Żeby ją odnaleźć, wykonuje on algorytm, który nazywamy „najkrótsza ścieżka”. Kupujesz coś w Internecie? Używasz wtedy (a przynajmniej należałoby!) bezpiecznej witryny sieciowej, która wykonuje algorytm szyfrowania. Gdy kupujesz jakieś towary w Sieci, kto Ci je dostarcza? Firma kurierska? Korzysta ona z algorytmów przydzielających paczki do furgonetek, a potem ustalających kolejność, w jakiej każdy kierowca powinien te paczki rozwozić. Algorytmy działają w komputerach na każdym kroku: w Twoim laptopie, smartfonie, na serwerach lub w systemach wbudowanych (np. w Twoim samochodzie czy w mikrofalówce, w systemach klimatyzacji) — wszędzie!

Co różni algorytmy wykonywane na komputerach od algorytmu wykonywanego przez Ciebie? Ty potrafisz tolerować algorytm, jeśli jest on niedokładnie opisany, a komputer nie. Na przykład, jeśli jedziesz do pracy, Twój algorytm jedź-do-pracy mógłby zawierać taką klauzulę: „jeśli jest tłoczno, wybierz inną trasę”. Ty możesz wiedzieć, co rozumiesz pod określeniem „jest tłoczno”, natomiast komputer tego nie pojmie.

Dlatego algorytm komputerowy jest zbiorem kroków prowadzących do wykonania zadania, opisanym na tyle precyzyjnie, że potrafi go wykonać komputer. Na czym polega ta precyzja, wiesz, jeśli masz choć trochę doświadczenia w programowaniu komputerów w Javie, C, C++, Pythonie, Fortranie, Mathlabie lub w czymś podobnym. Jeśli nie masz w dorobku ani jednego programu komputerowego, to niewykluczone, że poczujesz ten stopień dokładności, przyglądając się, jak opisuję algorytmy w tej książce.

Przejdźmy do następnego pytania: „Czego oczekujemy od algorytmu komputerowego?”.

[1] Lub, jak by rzekł mój kolega, z którym gram w hokeja: „What’s a nalgorithm?”.

Algorytmy komputerowe rozwiązują problemy obliczeniowe. Od algorytmu komputerowego oczekujemy dwóch rzeczy: mając dane do problemu, powinien zawsze wytwarzać poprawne rozwiązanie tego problemu, a robiąc to, powinien oszczędnie zużywać zasoby obliczeniowe. Przyjrzyjmy się obu tym postulatom.

Poprawność

Co oznacza poprawne rozwiązanie problemu? Na ogół potrafimy precyzyjnie określić, co powinno zawierać poprawne rozwiązanie. Na przykład, jeśli Twój GPS wytwarza poprawne rozwiązanie, gdy chodzi o znalezienie najlepszej trasy podróży, to mogłoby to polegać na wybraniu takiej trasy spośród wszystkich możliwych do wytyczenia między miejscem Twojego pobytu a miejscem docelowym, którą dotrzesz tam najszybciej. Albo trasy możliwie najkrótszej. Albo takiej, która nie tylko poprowadzi Cię do celu najszybciej, lecz również pozwoli uniknąć płacenia myta. Jasne, że informacje, których Twój GPS używa do wyznaczenia trasy, mogą nie odpowiadać rzeczywistości. O ile nie ma on dostępu w czasie rzeczywistym do danych o ruchu drogowym, mógłby przyjąć, że czas potrzebny do przebycia drogi równa się długości drogi podzielonej przez dopuszczalną na niej prędkość. Jeśli jednak droga jest zatłoczona, GPS mógłby Ci źle doradzić w poszukiwaniach najszybszej trasy. Mimo to nadal możemy uznać, że algorytm wytyczania trasy działa poprawnie — nawet jeśli nie można tego powiedzieć o jego danych wejściowych; dla zadanego wejścia algorytm trasujący określa trasę najszybszą.

Z kolei dla pewnych problemów ustalenie, czy dany algorytm wytwarza poprawne rozwiązanie, może się okazać trudne lub nawet niemożliwe. Jako przykład weźmy optyczne rozpoznawanie znaków. Czy ten obrazek o wymiarach 11×6 piksli jest cyfrą 5, czy literą S?

Niektórzy powiedzą, że to piątka, podczas gdy inni — że S, jak zatem moglibyśmy uznać poprawność lub niepoprawność decyzji komputerowej? Nie da się. W tej książce skoncentrujemy się na algorytmach komputerowych, których rozwiązania są znane.

Czasami jednak godzimy się z tym, że algorytm komputerowy produkuje niepoprawną odpowiedź, o ile tylko potrafimy panować nad tym, jak często mu się to zdarza. Dobry przykład stanowi szyfrowanie. Powszechnie używany kryptosystem RSA polega na określaniu, czy duże liczby — naprawdę duże, mające setki cyfr — są pierwsze. Jeśli zdarzyło Ci się pisać programy komputerowe, to jednym z nich był pewnie taki, który sprawdzał, czy liczba n jest pierwsza. Mógł on badać wszystkie potencjalne podzielniki od 2 do $n - 1$ i jeśli któryś z nich okazał się rzeczywiście podzielnikiem n, to n była liczbą złożoną. Jeśli żadna liczba między 2 a $n - 1$ nie jest podzielnikiem n, to n jest pierwsza. Jeśli jednak n ma setki cyfr, to potencjalnych podzielników jest mnóstwo — tak dużo, że nawet naprawdę szybki

komputer nie dałby rady tego sprawdzić w rozsądnym czasie. Oczywiście mógłbyś dokonać pewnych ulepszeń, na przykład wyeliminować wszystkie kandydatki parzyste po wykazaniu, że 2 nie jest podzielnikiem, lub poprzestać na dotarciu do $\sqrt{n}$ (bo jeśli d jest większe niż $\sqrt{n}$ i d jest podzielnikiem n, to n/d jest mniejsze niż $\sqrt{n}$ i również jest podzielnikiem n; jeśli więc n ma podzielnik, to znajdziesz go nim dojdziesz do $\sqrt{n}$). Jeśli n ma setki cyfr, to chociaż $\sqrt{n}$ ma około o połowę cyfr mniej niż n, nadal jest naprawdę wielką liczbą. I tu dobra wiadomość: znamy algorytm, który szybko sprawdza, czy liczba jest pierwsza. Złą wiadomością jest to, że może on popełniać błędy. W szczególności, jeśli oświadcza, że n jest złożona, to n jest na pewno złożona, lecz jeśli oświadcza, że n jest pierwsza, to istnieje pewna szansa, że n jest jednak złożona. Lecz owe złe wiadomości nie są aż tak złe: możemy utrzymywać wskaźnik błędu na naprawdę niskim poziomie, wynoszącym na przykład jeden błąd na każde 2^{50} razy. Jest to na tyle rzadko — jeden błąd raz na każde sto oktylionów — że większość z nas ze spokojem przyjmuje w RSA określanie tą metodą, czy liczba jest pierwsza.

Poprawność jest delikatnym zagadnieniem w innej klasie algorytmów, nazywanych aproksymacyjnymi. Algorytmy aproksymacyjne (przybliżania pewnej wartości — przyp. tłum.) są stosowane w problemach optymalizacyjnych, w których chcemy znaleźć najlepsze rozwiązanie względem pewnej miary ilościowej. Znajdowanie najszybszej trasy, wykonywane przez GPS, jest jednym z przykładów — tu miarą ilościową jest czas podróży. Dla niektórych problemów nie mamy dobrych algorytmów znajdujących optymalne rozwiązanie w rozsądnym czasie, znamy natomiast algorytm aproksymacyjny, który w zadowalającym czasie potrafi znaleźć rozwiązanie prawie optymalne. Przez „prawie optymalne" zazwyczaj rozumiemy, że ilościowa miara rozwiązania znajdowana przez algorytm aproksymacyjny różni się pewnym znanym czynnikiem od optymalnej miary ilościowej rozwiązania. Jeśli tylko określimy, ile ów pożądany czynnik wynosi, możemy mówić, że poprawnym rozwiązaniem algorytmu aproksymacyjnego jest każde rozwiązanie różniące się tym czynnikiem od rozwiązania optymalnego.

Użytkowanie zasobów

Co to znaczy w odniesieniu do algorytmu *efektywne użytkowanie zasobów obliczeniowych*? O jednej z miar efektywności wspomnieliśmy, omawiając algorytmy aproksymacyjne — był nią czas. Algorytm, który daje poprawne rozwiązanie, lecz zużywa dużo czasu na jego wytworzenie, może mieć znikomą wartość lub być bezwartościowy. Gdyby Twój GPS przez godzinę wyznaczał zalecaną trasę, chciałoby Ci się go w ogóle włączać? Rzeczywiście, czas jest podstawową miarą efektywności, której używamy do oceny algorytmu, kiedy już wykażemy, że daje poprawne rozwiązanie. Lecz nie jest to miara jedyna. Może nas interesować ilość pamięci komputerowej wymaganej przez algorytm („ślad" jaki odciska w pamięci), ponieważ algorytm musi działać w dostępnej pamięci. Inne zasoby, które mogą być używane przez algorytm, to: komunikacja sieciowa, losowe bity (ponieważ algorytmy, które dokonują

losowych wyborów, potrzebują źródła liczb losowych) lub operacje dyskowe (dla algorytmów przeznaczonych do pracy z danymi przechowywanymi na dysku).

W tej książce, tak jak w większości literatury o algorytmach, koncentrujemy się tylko na jednym zasobie — jest nim czas. Jak rozsądzamy o czasie wymaganym przez algorytm? W odróżnieniu od poprawności, niezależącej od konkretnego komputera, na którym działa algorytm, faktyczny czas algorytmów zależy od kilku czynników zewnętrznych względem samego algorytmu: szybkości komputera, języka programowania, w którym algorytm jest zrealizowany, kompilatora lub interpretera tłumaczącego program na kod wykonywany w komputerze, umiejętności osoby piszącej dany program i innych działań i zdarzeń zachodzących w komputerze równolegle z wykonywanym programem. Przy tym wszystkim zakłada się, że algorytm jest wykonywany tylko przez jeden komputer, mieszczący w pamięci wszystkie jego dane.

Gdybyśmy mieli oceniać szybkość algorytmu zrealizowanego w prawdziwym języku programowania, wykonując go na konkretnym komputerze dla określonych danych i mierząc jego czas działania, nie dowiedzielibyśmy się niczego o tym, jak szybko działałby ten algorytm z danymi innego rozmiaru, a może nawet z innymi danymi o tym samym rozmiarze. A gdybyśmy chcieli porównać względną szybkość danego algorytmu z innym, dla tego samego problemu, musielibyśmy zaimplementować oba i wykonać każdy z nich z różnymi danymi, o różnych rozmiarach. Jak zatem możemy ocenić szybkość algorytmu?

Otóż robimy to, łącząc dwa pomysły. Po pierwsze, określamy, jak długo działa algorytm w funkcji rozmiaru jego danych. W przykładzie ze znajdowaniem trasy dane wejściowe byłyby pewną reprezentacją mapy drogowej, a ich rozmiar zależałby od liczby skrzyżowań i liczby dróg łączących skrzyżowania na mapie. (Fizyczny rozmiar sieci dróg nie ma znaczenia, ponieważ wszystkie odległości możemy przedstawić za pomocą liczb, a wszystkie liczby zajmują tyle samo miejsca na wejściu; długość drogi nie wpływa na rozmiar danych wejściowych). W prostszym przykładzie — przeszukiwaniu zadanej listy elementów w celu sprawdzenia, czy określony element jest na niej obecny — rozmiarem danych byłaby liczba pozycji na liście.

Po drugie, skupiamy się na tym, jak szybko funkcja charakteryzująca czas działania rośnie z rozmiarem danych wejściowych, tzn. na *tempie wzrostu* czasu działania. W rozdziale 2 zapoznamy się z notacją, której używamy do scharakteryzowania czasu działania algorytmu, lecz to, co jest w naszym podejściu najbardziej interesujące, to uwzględnianie w czasie działania tylko dominującego składnika; nie zwracamy przy tym uwagi na współczynniki. Koncentrujemy się zatem na **rzędzie wzrostu** czasu działania. Załóżmy na przykład, że udało się nam ustalić, iż dana realizacja pewnego algorytmu przeszukiwania listy n elementów zajmuje $50n + 125$ cykli maszynowych. Składnik $50n$ zdominuje składnik 125, jeśli n stanie się dostatecznie duże, poczynając od $n \geq 3$ i zwiększając jeszcze tę przewagę dla list o większych rozmiarach. Wobec tego, opisując czas działania tego hipotetycznego algorytmu, nie bierzemy pod uwagę składnika 125 małego rzędu. Może się zdziwisz, ale odrzucamy też współczynnik 50. Prowadzi to do określenia czasu działania jako rosnącego liniowo

z rozmiarem danych n. Oto inny przykład: gdyby algorytm zużywał $20n^3 + 100n^2 + 300n + 200$ cykli maszynowych, to powiedzielibyśmy, że jego czas działania rośnie jak n^3. Również w tym przypadku składniki niższego rzędu: $100n^2$, $300n$ i 200, stają się coraz mniej istotne ze wzrostem rozmiaru n danych wejściowych.

W praktyce lekceważone przez nas współczynniki mają jednak znaczenie. Zależą one jednak w tak dużym stopniu od czynników zewnętrznych, że jest prawie pewne, iż przy porównywaniu dwóch algorytmów A i B, mających ten sam rząd wzrostu i działających na tych samych danych, A może działać szybciej niż B dla pewnej kombinacji maszyny, języka programowania, kompilatora (lub interpretera) i programisty, a B zadziała szybciej niż A dla innej kombinacji. Oczywiście, jeżeli oba algorytmy A i B produkują poprawne rozwiązania i A zawsze działa dwa razy szybciej niż B, to jeśli wszystko inne jest takie samo, będziemy zawsze preferowali wykonywanie A zamiast B. Jednakże z punktu widzenia abstrakcyjnego porównywania algorytmów zawsze koncentrujemy się na rzędzie wzrostu, nie przystrajając go współczynnikami lub składnikami niskiego rzędu.

I ostatnie pytanie, które stawiamy w tym rozdziale: „Dlaczego miałoby mi się opłacać zajmowanie algorytmami komputerowymi?". Odpowiedź na nie zależy od tego, kim jesteś.

Algorytmy komputerowe dla niekomputerowców

Nawet jeśli w sferze komputerów nie uważasz się za osobę dobrze poinformowaną, algorytmy komputerowe stanowią dla Ciebie dużą wartość. W końcu, wyjąwszy sytuację, w której podróżujesz po bezdrożach bez GPS-u, zapewne używasz go co dnia. Szukałeś czegoś dzisiaj w Internecie? W używanej przez Ciebie wyszukiwarce — Google, Bing lub jakiejkolwiek innej — są stosowane wyrafinowane algorytmy przeszukiwania Sieci i rozstrzygania, w jakiej kolejności należy przedstawiać ich wyniki. Prowadziłeś dziś samochód? O ile nie jeździsz klasycznym wehikułem, jego komputery pokładowe podjęły podczas Twej podróży miliony decyzji, a wszystkie oparte były na algorytmach. Mógłbym tak wymieniać bez końca.

Jako docelowy użytkownik algorytmów sobie zawdzięczasz chęć dowiedzenia się czegoś o tym, jak projektujemy, charakteryzujemy i oceniamy algorytmy. Zakładam, że przejawiasz przynajmniej umiarkowane zainteresowanie, skoro ta książka trafiła do Twoich rąk i czytasz ją do tego miejsca. Powodzenia! Zobaczmy, czy uda się nam rozkręcić Cię na tyle, że na następnym spotkaniu towarzyskim dotrzymasz tonu innym, gdy rozmowa zejdzie na algorytmy[2].

[2] No tak, przyznaję, jeśli nie mieszkasz w Dolinie Krzemowej, temat algorytmów rzadko wypłynie podczas koktajli, w których bierzesz udział, niemniej z pewnych względów my, profesorowie informatyki, uważamy, że jest ważne, aby nasi studenci nie konsternowali nas na spotkaniach towarzyskich brakiem wiedzy z poszczególnych dziedzin informatyki.

Algorytmy komputerowe dla komputerowców

Jeśli jesteś za pan brat z komputerami, to masz większe obowiązki wobec algorytmów! One nie tylko znajdują się w centrum wszystkiego, co się dzieje w Twoim komputerze — algorytmy są technologią, i to równie istotną jak wszystko, co współtworzy Twój komputer. Możesz zapłacić dodatkową cenę za komputer z najnowszym i największym procesorem, lecz aby te pieniądze okazały się trafionym wydatkiem, potrzebujesz, by w tym komputerze działały realizacje dobrych algorytmów.

Oto przykład ilustrujący, że algorytmy rzeczywiście stanowią technologię. W trzecim rozdziale przyjrzymy się trzem różnym algorytmom sortowania w porządku rosnącym listy n wartości. Niektóre z tych algorytmów będą osiągały czasy działania rosnące jak n^2, lecz inne będą działać w czasie rosnącym tylko jak $n \lg n$. Co to jest $\lg n$? Jest to logarytm przy podstawie 2 z n, czyli $\log_2 n$. Informatycy używają logarytmów z podstawą 2 tak często, że podobnie jak matematycy i naukowcy, którzy dla skrócenia stosują zapis $\ln n$ na oznaczenie logarytmu naturalnego — $\log_e n$, również oni używają własnego skrótu dla logarytmów przy podstawie 2. I tak, ponieważ funkcja $\lg n$ jest odwrotnością funkcji wykładniczej, rośnie ona bardzo powoli z n. Jeśli $n = 2^x$, to $x = \lg n$. Na przykład $2^{10} = 1024$, więc $\lg 1024$ wynosi tylko 10, podobnie $2^{20} = 1\,048\,576$, zatem $\lg 1\,048\,576$ wynosi zaledwie 20, a $2^{30} = 1\,073\,741\,824$, co oznacza, że $\lg 1\,073\,741\,824$ wynosi raptem 30. Tak więc we wzroście $n \lg n$ w porównaniu ze wzrostem n^2 zachodzi wymiana czynnika n na jedynie $\ln n$, a to już jest gra warta świeczki.

Skonkretyzujmy nieco bardziej ten przykład, wystawiając szybszy komputer (komputer A) wykonujący algorytm sortowania, którego czas działania dla n wartości rośnie jak n^2, przeciw wolniejszemu komputerowi B, wykonującemu algorytm sortowania w czasie rosnącym jak $n \lg n$. Każdy z komputerów musi posortować 10 milionów liczb. (Choć może się wydawać, że 10 milionów liczb to dużo, jeśli są to 8-bajtowe liczby całkowite, ich rozmiar na wejściu zajmie około 80 megabajtów, co zmieści się w pamięci nawet niedrogiego laptopa sprzed wielu lat). Przypuśćmy, że komputer A wykonuje 10 miliardów operacji na sekundę (jest szybszy od każdego sekwencyjnego komputera w chwili, gdy piszę te słowa), a komputer B wykonuje tylko 10 milionów operacji na sekundę, czyli komputer A jest 1000 razy szybszy od komputera B, biorąc pod uwagę samą moc obliczeniową. Aby powiększyć jeszcze różnicę, załóżmy, że światowej sławy programistka koduje komputer A w języku maszynowym, a kod wynikowy wymaga do posortowania n liczb $2n^2$ rozkazów. Załóżmy dalej, że program dla komputera B pisze programista zupełnie przeciętny, korzystając z języka wysokiego poziomu i mało efektywnego kompilatora, w wyniku czego powstaje kod złożony z $50n \lg n$ rozkazów. Aby posortować 10 milionów liczb, komputer A zużywa

$$\frac{2 \cdot (10^7)^2 \text{ rozkazów}}{10^{10} \text{ rozkazów / sekundę}} = 20\,000 \text{ sekund},$$

co stanowi więcej niż 5 i pół godziny, podczas gdy komputerowi B zabiera to

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ rozkazów}}{10^7 \text{ rozkazów / sekundę}} \approx 1163 \text{ sekundy}$$

— równowartość mniej niż 20 minut. Stosując algorytm, którego czas działania rośnie znacznie wolniej, nawet z marnym kompilatorem, komputer B działa 17 razy szybciej niż komputer A! Przewaga algorytmu $n \lg n$ byłaby jeszcze wyraźniejsza, gdybyśmy sortowali 100 milionów liczb. Komputerowi A zabrałoby to ponad 23 dni, algorytm $n \lg n$ na komputerze B uporałby się z robotą w cztery godziny. Uogólniając, wraz ze wzrostem rozmiaru problemu rośnie względna przewaga algorytmu $n \lg n$.

Nawet mimo imponującego postępu, który nieustannie obserwujemy w sprzęcie komputerowym, całościowa produktywność systemu zależy w równym stopniu od doboru efektywnych algorytmów, co od doboru szybkiego sprzętu lub sprawnie działających systemów operacyjnych. Błyskawiczny postęp, jaki dokonał się w innych technologiach komputerowych, dotyczy także algorytmów.

Co czytać dalej

W moim bardzo subiektywnym odczuciu najklarowniejszym i najbardziej użytecznym źródłem wiedzy o algorytmach komputerowych jest *Wprowadzenie do algorytmów* [CLRS09], napisane przez czterech diabelnie przystojnych facetów. Książka ta jest powszechnie określana jako „CLRS", od inicjałów autorów. Sięgałem do niej po większość materiału pomieszczonego w tej książce. Jest ona nieporównanie bardziej kompletna od tej książki, lecz zakłada się w niej, że masz choć trochę doświadczenia w programowaniu komputerów i jesteś za pan brat z matematyką. Jeśli uznasz, że odpowiada Ci poziom matematyczny niniejszej książki, i masz odwagę ruszyć głębiej w temat, nie możesz postąpić lepiej, niż sięgnąć po CLRS. (W mojej skromnej opinii, ma się rozumieć).

Książka Johna MacCormica *Nine Algorithms That Changed the Future* [Mac12] zawiera opis kilku algorytmów i związanych z nimi kwestii obliczeniowych, które oddziałują na nasze codzienne życie. Ujęcie zastosowane przez MacCormica jest mniej techniczne niż w tej książce. Jeśli dojdziesz do wniosku, że tutaj było „zbyt matematycznie", to polecam Ci skosztowanie lektury MacCormica. Zdołasz przyswoić z niej wiele, nawet jeśli masz wątłe przygotowanie matematyczne.

W mało prawdopodobnym przypadku, gdy uznasz, że CLRS jest zbyt rozwodnione, możesz spróbować zaczerpnąć z wielotomowego zbioru Donalda Knutha *The Art of Computer Programming* [Knu97, Knu98a, Knu98b, Knu11]. Chociaż tytuł serii sugeruje, że skupia się ona na pisaniu kodu, książki te zawierają wspaniałą, głęboką analizę algorytmów[3]. Ostrzegam jednak: materiał zawarty w *TAOCP* sięga głęboko. Przy okazji, jeśli zastanawiasz się, skąd się wzięło słowo „algorytm", to Knuth powiada, że pochodzi ono od imienia „al-Khowârizmî" perskiego matematyka z IX wieku.

[3] *Sztuka programowania*, WNT, Warszawa 2002 – 2007. Jak dotąd ukazały się 4 książki z jeszcze nie ukończonego cyklu, który Donald Knuth zaplanował na 7 tomów — *przyp. tłum.*

Oprócz CLRS przez lata ukazało się kilka innych świetnych tekstów o algorytmach komputerowych. Noty do rozdziału 1 w CLRS zawierają odwołania do wielu takich tekstów. Zamiast powielać je tutaj, kieruję Cię wprost do CLRS.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

DOWIEDZ SIĘ, JAK DZIAŁAJĄ APLIKACJE KOMPRESUJĄCE I SZYFRUJĄCE!

Każdy program działa według określonego algorytmu — Twoja nawigacja GPS, system płatności elektronicznych, wyszukiwarka Google. Algorytmy są jak przepisy kucharskie: zrób to, sprawdź tamto. Jednak konsekwencje popełnienia błędu w algorytmie są zupełnie inne niż w przypadku niesprawdzonego przepisu. To właśnie algorytmy decydują o czasie wykonania skomplikowanych operacji przez programy komputerowe, a ich odpowiednia lub nieodpowiednia implementacja może sprawić, że Twój projekt wart miliony odniesie sukces lub poniesie porażkę.

Dzięki tej książce będziesz mógł bezboleśnie wkroczyć w świat algorytmów. W trakcie lektury dowiesz się, czym są algorytmy, jak się je projektuje i prezentuje. Po wstępie teoretycznym poznasz najpopularniejsze algorytmy sortowania i wyszukiwania, algorytmy znajdowania najkrótszej ścieżki oraz algorytmy operujące na ciągach znaków. Następnie przejdziesz do najciekawszych zagadnień związanych z kryptografią i kompresją danych. Zastanawiasz się, czy są miejsca, w których znane algorytmy nie radzą sobie zbyt dobrze? To problemy NP-zupełne — z nimi też będziesz mógł się zaznajomić. Ta książka jest interesującym przewodnikiem po świecie algorytmów, a zarazem przyjemną lekturą dla każdego programisty i pasjonata informatyki.

Poznaj algorytmy:

- sortujące i wyszukujące
- znajdowania najkrótszej ścieżki
- kryptograficzne
- kompresujące

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

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-6736-4
9 788328 367364

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 54,90 zł