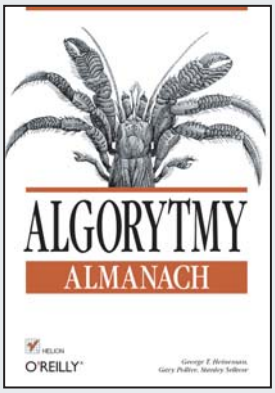
Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Algorytmy. Almanach



Autorzy: George Heineman, Gary Pollice, Stanley Selkow
Tłumaczenie: Zdzisław Płoski
ISBN: 978-83-246-2209-2
Tytuł oryginału: Algorithms in a Nutshell
Format: 168×237, stron: 352

Cała wiedza o algorytmach w jednym podręczniku!

- Jaki wpływ na różne algorytmy wywierają podobne decyzje projektowe?
- Jak rozwiązywać problemy dotyczące kodowania?
- Jak wykorzystać zaawansowane struktury danych do usprawnienia algorytmów?

Tworzenie niezawodnego oprogramowania wymaga stosowania sprawnych algorytmów. Jednak programiści rzadko poświęcają im uwagę, dopóki nie pojawią się kłopoty. Aby ich uniknąć, powinieneś wiedzieć, w jaki sposób poprawianie efektywności najważniejszych algorytmów przesądza o sukcesie Twoich aplikacji. W tej książce znajdziesz przetestowane i wypróbowane metody wykorzystywania oraz poprawiania skuteczności algorytmów – do użycia w celu wdrożenia sprawnych rozwiązań programistycznych.

Książka „Algorytmy. Almanach" to cała wiedza o algorytmach, potrzebna ambitnemu programiście, zebrana w jeden kompletny podręcznik. Książka zawiera opisy algorytmów do rozwiązywania rozmaitych problemów, pomaga w wyborze i realizacji algorytmów odpowiednich do Twoich potrzeb, a także dostarcza wydajnych rozwiązań zakodowanych w kilku językach programowania, które łatwo można zaadaptować w konkretnych zadaniach. Dzięki temu podręcznikowi nauczysz się projektować struktury danych, a także dowiesz się, na czym polega przeszukiwanie drzewa binarnego oraz jak korzystać z informacji heurystycznych. Poznasz zaawansowane struktury danych, przydatne do usprawniania algorytmów, a jednocześnie niezbędne dla zagwarantowania pełnego sukcesu Twoich rozwiązań programistycznych.

- Algorytmy w ujęciu matematycznym
- Wzorce i dziedziny
- Algorytmy sortowania
- Wyszukiwanie sekwencyjne
- Przeszukiwanie drzewa binarnego
- Algorytmy grafowe
- Drzewa poszukiwań
- Korzystanie z informacji heurystycznych
- Algorytmy przepływu w sieciach
- Geometria obliczeniowa
- Zapytania przedziałowe

Cała wiedza o algorytmach, potrzebna każdemu programiście!

Spis treści

ROZDZIAŁ 2.

Algorytmy w ujęciu matematycznym

Wybierając algorytm do rozwiązania problemu, próbujesz przewidzieć, który będzie najszybszy dla konkretnego zbioru danych na konkretnej platformie (lub rodzinie platform). Określenie oczekiwanego czasu działania danego algorytmu jest procesem z natury matematycznym. W tym rozdziale przedstawiamy narzędzia matematyczne pomagające w takich przewidywaniach. Po przeczytaniu tego rozdziału Czytelnicy będą rozumieć różne pojęcia matematyczne występujące w tej książce.

Wspólnym motywem tego rozdziału (w istocie — przyjętym w całej książce) jest założenie, że wszystkie hipotezy i przybliżenia mogą się różnić o pewną stałą, przy czym w naszych uogólnieniach stałe takie możemy pomijać. Dla wszystkich algorytmów ujętych w tej książce stałe te są małe w przypadku niemal wszystkich platform.

Rozmiar konkretnego problemu

Przez egzemplarz (ukonkretnienie) problemu rozumie się określony zbiór danych, na którym działa program. W większości problemów czas wykonania programu wzrasta z rozmiarem kodowania rozwiązywanego egzemplarza. Z drugiej strony reprezentacje nazbyt zwarte (np. powstałe z użyciem technik kompresji) mogą niepotrzebnie spowalniać wykonanie programu. Zdefiniowanie optymalnego sposobu zakodowania konkretnego problemu jest zaskakująco trudne, ponieważ problemy występują w świecie rzeczywistym i trzeba je tłumaczyć na odpowiednią reprezentację maszynową, aby można było je rozwiązać na komputerze. Rozważmy dwa zakodowania liczby x, pokazane nieco dalej w ramce „Konkrety są kodowane".

O ile to tylko możliwe, chcemy w ocenie algorytmów korzystać z założenia, że zakodowanie konkretnego problemu nie ma decydującego wpływu na możliwość uzyskania sprawnej realizacji algorytmu. Choć rozmiary obu kodowań (w ramce) są niemal identyczne, wiąże się z nimi różna sprawność podstawowej operacji, zależna od tego, czy x ma w reprezentacji dwójkowej parzystą, czy nieparzystą liczbę bitów o wartości 1.

Wybór reprezentacji konkretnego problemu zależy od typu i rozmaitości operacji, które mają być wykonywane. Projektowanie efektywnych algorytmów często zaczyna się od wyboru odpowiednich struktur danych, za pomocą których można przedstawić problem wymagający rozwiązania, co pokazano na rysunku 2.1.

Rysunek 2.1. Bardziej skomplikowane zakodowanie konkretnego problemu

Konkrety są kodowane

Załóżmy, że masz wielką liczbę x i chcesz wyznaczyć parzystość liczby jedynek w jej reprezentacji dwójkowej (to znaczy, chcesz sprawdzić, czy w jej reprezentacji dwójkowej występuje parzysta liczba bitów o wartości 1). Jeśli na przykład x = 15 137 300 128, to jej przedstawienie przy podstawie 2 jest następujące:

x_2= 11100001100100000011011111010100000

a liczba jedynek jest w niej parzysta. Rozważamy dwie możliwości kodowania:

Pierwsze kodowanie x: 11100001100100000011011111010100000

W tym przypadku reprezentacją problemu jest 34-bitowe przedstawienie x przy podstawie 2, zatem rozmiar wejścia[1] wynosi n = 34. Zauważmy, że $log_2(x) \cong 33{,}82$, zatem kodowanie to jest optymalne. Aby jednak obliczyć parzystość liczby jedynek, trzeba sprawdzić każdy bit. Optymalny czas obliczenia parzystości rośnie liniowo wraz z n (logarytmicznie z x).

Liczbę x można też zakodować w postaci n-bitowej z dodaniem bitu sumy kontrolnej, który wskazuje parzystość liczby jedynek w kodowaniu x.

Drugie kodowanie x: 11100001100100000011011111010100000[0]

Ostatni bit x w drugim kodowaniu jest równy 0, co oznacza, że x ma parzystą liczbę jedynek (parytet parzysty[2] = 0). W tej reprezentacji n = 35. W obu przypadkach rozmiar n zakodowanego egzemplarza problemu rośnie logarytmicznie ze wzrostem x. Jednak czas optymalnego algorytmu obliczenia parzystości x z użyciem pierwszego kodowania rośnie logarytmicznie z rozmiarem kodowania x, a w drugim kodowaniu czas algorytmu optymalnego jest stały i nie zależy od rozmiaru kodowania x.

[1] Tam, gdzie nie powoduje to niejednoznaczności, używamy określeń *wejście*, *wyjście* zamiast *dane wejściowe*, *dane wyjściowe* — *przyp. tłum.*

[2] Albo „równą parzystość" (ang. *even parity*); wskutek tłumaczenia w matematyce, fizyce i informatyce angielskiego *parity* jako „parzystość" powstaje tu zabawna sytuacja: dosł. „parzystość parzysta" — *przyp. tłum.*

Ponieważ nie możemy formalnie zdefiniować rozmiaru konkretnego problemu, zakładamy, że jest on zakodowany w jakiś ogólnie akceptowany sposób. Na przykład, sortując *n* liczb, przyjmujemy ogólną zasadę, że każda z *n* liczb mieści się w słowie danego komputera, a rozmiar konkretnych danych, które mają być posortowane, wynosi *n*. Gdyby któraś z tych liczb wymagała więcej niż jednego słowa — lecz tylko skończonej, ustalonej liczby słów — to nasza miara rozmiaru konkretnego problemu różniłaby się o pewną stałą. Tak więc algorytm, który wykonuje obliczenia na 64-bitowych liczbach całkowitych, może działać dwa razy dłużej niż podobny algorytm zakodowany dla liczb przechowywanych na 32 bitach.

Do przechowywania zestawów (kolekcji) informacji większość języków programowania udostępnia tablice — ciągłe obszary pamięci indeksowane całkowitym *i*, umożliwiające szybki dostęp do *i*-tego elementu. Tablica jest jednowymiarowa, jeśli każdy element mieści się w słowie platformy (np. tablica liczb całkowitych, wartości boolowskich lub znaków)[3]. Inne tablice mają wiele wymiarów, umożliwiając ciekawsze reprezentacje danych, jak pokazano na rysunku 2.1. Trzeba też dodać, co pokazano nieco dalej, w ramce „Wpływ kodowania na sprawność", że kodowanie może oddziałać na sprawność algorytmu.

Wskutek olbrzymiej różnorodności języków programowania i sprzętu komputerowego, na którym są wykonywane programy, badacze algorytmów godzą się z tym, że nie są w stanie obliczyć z wyśrubowaną dokładnością kosztów wynikających z użycia takiego czy innego kodowania w implementacji. Przyjmują więc, że koszty działania różniące się stałym mnożnikiem są asymptotycznie równoważne. Choć taka definicja byłaby niepraktyczna w realnych sytuacjach (kto byłby zadowolony, słysząc, że musi zapłacić rachunek tysiąckrotnie wyższy niż zakładany?), służy uniwersalnie do porównywania algorytmów. Jest oczywiste, że gdy chodzi o zrealizowanie algorytmu w kodzie na potrzeby przemysłowe, szczegóły wyrażone w tych stałych są istotne.

Tempo rośnięcia funkcji

Powszechnie akceptowaną metodą opisywania zachowania algorytmu jest przedstawienie tempa wzrostu czasu jego wykonania w funkcji rozmiaru konkretnego, wejściowego problemu. Charakteryzowanie w ten sposób sprawności algorytmu jest abstrakcją, w której pomija się szczegóły. Właściwe posługiwanie się tą miarą wymaga uświadomienia sobie szczegółów, które taka abstrakcja skrywa.

Każdy program jest wykonywany na *platformie* — termin ten ogólnie określa, co następuje:

- komputer, na którym program jest wykonywany, jego jednostkę centralną (CPU), pamięć podręczną, jednostkę obliczeń zmiennopozycyjnych (FPU) i inne właściwości zrealizowane układowo;
- język programowania, w którym program jest napisany, wraz z kompilatorem lub interpreterem i ustawieniami dotyczącymi optymalizowania generowanego kodu;
- system operacyjny;
- inne procesy wykonywane w tle.

[3] Jest to dość tendencyjna definicja, ponieważ liczba rozmiarów tablicy jest wyznaczona przez liczbę wskaźników (adresów pośrednich) potrzebnych do dotarcia do jej elementu. Sam element tablicy może zajmować wiele słów maszynowych lub część słowa — *przyp. tłum.*

Wpływ kodowania na sprawność

Załóżmy, że program przechowuje informacje o układzie okresowym pierwiastków. Do częstych pytań należą: (a) „Ile wynosi ciężar atomowy pierwiastka o numerze *N*?", (b) „Ile wynosi ciężar atomowy pierwiastka o nazwie *X*?" i (c) „Jak się nazywa pierwiastek o numerze *N*?". Ciekawą kwestią w tym zadaniu jest to, że według danych na styczeń 2008 r. pierwiastek 117 nie został odkryty, choć pierwiastek 118, Ununoctium, jest już znany.

Pierwsze kodowanie układu okresowego. Zapamiętaj dwie tablice: `nazwaPierwiastka[]`, której *i*-ta wartość jest nazwą pierwiastka o liczbie atomowej *i*, i tablicę `ciężarElementu`[4], której *i*-ty element ma wartość ciężaru atomowego pierwiastka.

Drugie kodowanie układu okresowego. Zapamiętaj napis złożony z 2626[5] znaków, reprezentujący cały układ okresowy. Pierwsze 62 znaków wygląda następująco:

1 H Hydrogen 1,00794

2 He Helium 4,002602

3 Li Lithium 6,941

W poniższej tabeli pokazano wyniki z 32 prób liczących po 100 000 losowych zapytań (w tym również niepoprawnych). Usunięto wynik najlepszy i wynik najgorszy, zostawiając 30 prób, których średni czas wykonania (i odchylenie standardowe) podano w milisekundach.

	Ciężar	**Numer**	**Nazwa**
Kodowanie 1	2,1±5,45	131,73±8,83	2,63±5,99
Kodowanie 2	635,07±41,19	1050,43±75,60	664,13±45,90

Zgodnie z oczekiwaniami drugie kodowanie daje gorsze czasy wykonania, ponieważ każde zapytanie wymaga działań na napisach. Kodowanie 1 umożliwia sprawne przetwarzanie zapytań o *ciężar* i *nazwę*, lecz pytania o *numer* zmuszają do przeszukania nieuporządkowanego układu.

Ten przykład wykazuje, że różne zakodowanie danych powoduje (lub może powodować) olbrzymie różnice w czasie wykonania. Pokazuje również, że projektanci powinni zastanowić się nad wyborem operacji, które chcą optymalizować.

Przyjmuje się tu założenie, że zmiana któregokolwiek z parametrów platformy zmieni czas wykonania programu o czynnik stały. Aby osadzić dalsze omówienie w jakimś konkretnym kontekście, dokonamy zwięzłego przeglądu algorytmu WYSZUKIWANIA LINIOWEGO, przedstawionego w rozdziale 5. WYSZUKIWANIE LINIOWE polega na przejrzeniu po jednym wykazu $n \geq 1$ określonych elementów aż do znalezienia potrzebnej wartości v. Na razie załóżmy, że:

- na wykazie jest wyodrębnionych n elementów,
- poszukiwany element v występuje na wykazie,
- każdy element wykazu może mieć wartość v z jednakowym prawdopodobieństwem.

Aby zrozumieć osiągi WYSZUKIWANIA LINIOWEGO, musimy wiedzieć, ile elementów jest w nim sprawdzanych „średnio". Ponieważ wiadomo, że v występuje na wykazie i każdy element

[4] W nazwach nie należących do większych fragmentów kodu dla wygody czytania używamy liter ze znakami diakrytycznymi. Nazwy takie są również w tekście odmieniane przez przypadki — *przyp. tłum.*

[5] Ta liczba odnosi się do angielskich nazw pierwiastków — *przyp. tłum.*

może przyjąć wartość v z równym prawdopodobieństwem, średnia liczba $E(n)$ elementów sprawdzanych jest sumą liczb elementów sprawdzanych dla każdej z n wartości, podzieloną przez n. Wyrażając to matematycznie:

$$E(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{1}{2}n + \frac{1}{2}$$

Zatem zgodnie z tymi założeniami w WYSZUKIWANIU LINIOWYM jest sprawdzanych około połowy elementów na wykazie n różnych elementów. Jeśli liczba elementów wykazu zostanie podwojona, to WYSZUKIWANIE LINIOWE powinno sprawdzać około dwa razy więcej elementów; oczekiwana liczba badań jest liniową funkcją n. Oczekiwana liczba badań jest zatem *liniowa*, czyli wynosi „około" $c \cdot n$, gdzie c jest pewną stałą. W danym wypadku $c = 1/2$. Podstawowy wniosek z tej analizy jest taki, że stała c jest nieistotna dla dłuższych wykonań, ponieważ najważniejszym czynnikiem kosztu jest rozmiar egzemplarza problemu, czyli n. Wraz ze wzrostem n błąd w stwierdzeniu, że

$$\frac{1}{2}n \approx \frac{1}{2}n + \frac{1}{2}$$

staje się mało znaczący. W rzeczywistości proporcja między oboma stronami tego przybliżenia dąży do 1. To znaczy:

$$\lim_{n\to\infty} = \frac{\left(\frac{1}{2}n\right)}{\left(\frac{1}{2}n + \frac{1}{n}\right)} = 1$$

aczkolwiek błąd w oszacowaniu jest znaczący dla małych wartości n. Biorąc to pod uwagę, mówimy, że tempo wzrostu oczekiwanej liczby elementów, które trzeba zbadać w WYSZUKIWANIU LINIOWYM, jest liniowe. Tym samym pomijamy czynnik stały i koncentrujemy się tylko na egzemplarzach problemów dużego rozmiaru.

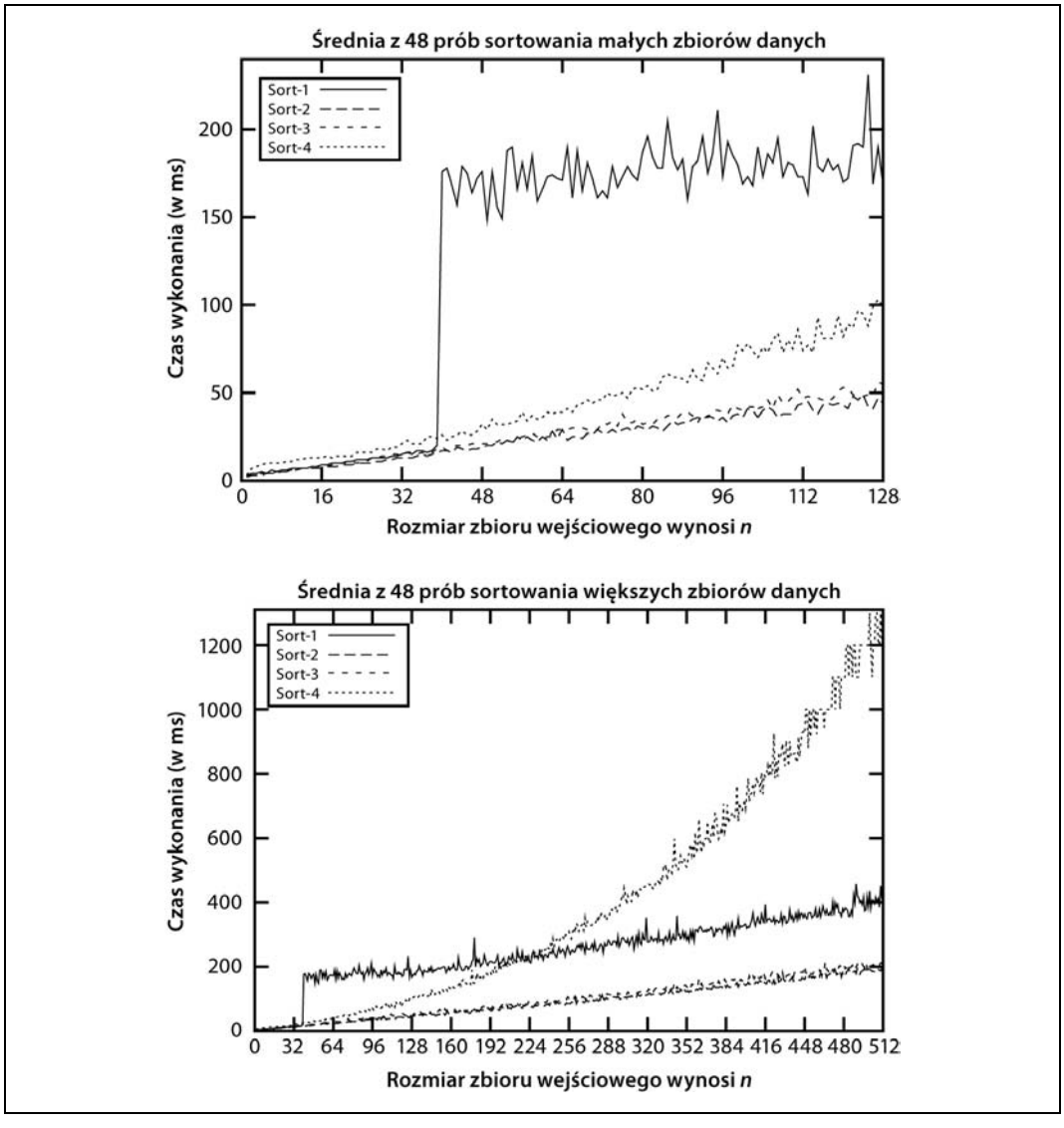
Posługując się przy wybieraniu algorytmów abstrakcją tempa wzrostu, musimy być świadomi następujących założeń:

Stałe są ważne
: Z tego powodu używamy superkomputerów i unowocześniamy nasze komputery co pewien czas.

Rozmiar n nie zawsze jest duży
: W rozdziale 4. zobaczymy, że czas wykonania algorytmu QUICKSORT rośnie wolniej niż czas wykonania SORTOWANIA PRZEZ WSTAWIANIE. Mimo to SORTOWANIE PRZEZ WSTAWIANIE przewyższa QUICKSORT dla małych tablic na tej samej platformie.

Tempo wzrostu czasu algorytmu przesądza o tym, jak będzie się on zachowywał w przypadku coraz większych konkretnych problemów. Odnieśmy tę podstawową zasadę do bardziej złożonego przykładu.

Zajmiemy się oceną czterech algorytmów sortowania w pewnym zadaniu sortowania. Następujące dane dotyczące sprawności wygenerowano podczas sortowania bloku n losowych napisów. Dla bloków długości $n = 1…512$ wykonano 50 prób. Usunięto sprawność najlepszą i najgorszą i na wykresie przedstawionym na rysunku 2.2 ukazano średni czas obliczenia (w mikrosekundach) pozostałych 48 wyników. Wariancja między tymi wykonaniami jest zaskakująca.



Rysunek 2.2. Porównanie czterech algorytmów sortowania małych zbiorów danych

Jedna z możliwych interpretacji tych wyników polega na próbie obmyślenia funkcji, która przewidywałaby sprawność każdego algorytmu na egzemplarzu problemu o rozmiarze *n*. Ponieważ odgadnięcie takiej funkcji jest mało prawdopodobne, korzystamy z dostępnego na rynku oprogramowania do obliczania krzywej trendu za pomocą procesu statystycznego zwanego analizą regresji. Miarą „dopasowania" krzywej trendu do rzeczywistych danych jest liczba z przedziału [0, 1], nazywana wartością R^2. Wartości bliskie 1 wskazują duże dopasowanie. Jeśli na przykład $R^2 = 0{,}9948$, to szansa na to, że dopasowanie krzywej trendu jest spowodowane losowymi zmianami w danych, wynosi tylko 0,52%.

SORT-4 działa wyraźnie najgorzej z tych algorytmów. Dla 512 punktów danych naniesionych na wykresie krzywa trendu, której odpowiadają te dane, wygląda następująco:

$$y = 0{,}0053 \cdot n^2 - 0{,}3601 \cdot n + 39{,}212$$

$$R^2 = 0{,}9948$$

Poziom ufności R^2 tak bliski 1 zaświadcza, że oszacowanie to jest dokładne. Najszybszą realizację w zadanym przedziale punktów umożliwia algorytm SORT-2. Jego zachowanie jest scharakteryzowane przez następujące równanie:

$$y = 0{,}05765 \cdot n \cdot \log(n) + 7{,}9653$$

SORT-2 z początku minimalnie przewyższa SORT-3, ostatecznie zaś można przyjąć, że jest o 10% szybszy niż SORT-3. Algorytm SORT-1 zachowuje się dwojako. Dla bloków 39 napisów — lub mniejszych — zachowanie jest opisane wzorem:

$$y = 0{,}0016 \cdot n^2 - 0{,}2939 \cdot n + 3{,}1838$$

$$R^2 = 0{,}9948$$

Jednakże dla 40 napisów i więcej jego zachowanie jest scharakteryzowane przez:

$$y = 0{,}0798 \cdot n \cdot \log(n) + 142{,}7818$$

Współczynniki liczbowe w tych równaniach całkowicie zależą od *platformy*, na której działają dane realizacje. Jak opisano wcześniej, tego rodzaju przypadkowe różnice nie mają znaczenia. Trend długoterminowy, towarzyszący wzrostowi *n*, dominuje w obliczeniach tego typu zachowań. I rzeczywiście, na rysunku 2.2 zachowanie uwidoczniono w dwu różnych przedziałach, aby pokazać, że dopóki *n* nie jest dostatecznie duże, dopóty prawdziwe zachowanie algorytmu może się nie ujawnić.

Osoby projektujące algorytmy dążą do zrozumienia różnic w zachowaniu poszczególnych algorytmów. Kod takich algorytmów można pobrać z powszechnie dostępnych magazynów („repozytoriów") kodu źródłowego, a obejrzenie wpływu wyborów projektowych na ogólne działanie jest bardzo pouczające. SORT-1 odzwierciedla działanie funkcji `qsort` w systemie Linux 2.6.9. Przeglądając kod źródłowy (można go znaleźć w dowolnej składnicy kodu linuksowego[6]), napotykamy następujący komentarz: „Qsort routine from Bentley & McIlroy's Engineering a Sort Function". Bentley & McIlroy [1993] podają, jak zoptymalizować QUICKSORT, zmieniając strategię stosownie do rozmiarów problemu: mniejszych niż 7, zawartych między 8 a 39 i większych od 40. Przyjemnie jest skonstatować, że przedstawione tutaj wyniki empiryczne potwierdzają to, co zawiera implementacja.

Analiza przypadku najlepszego, średniego i najgorszego

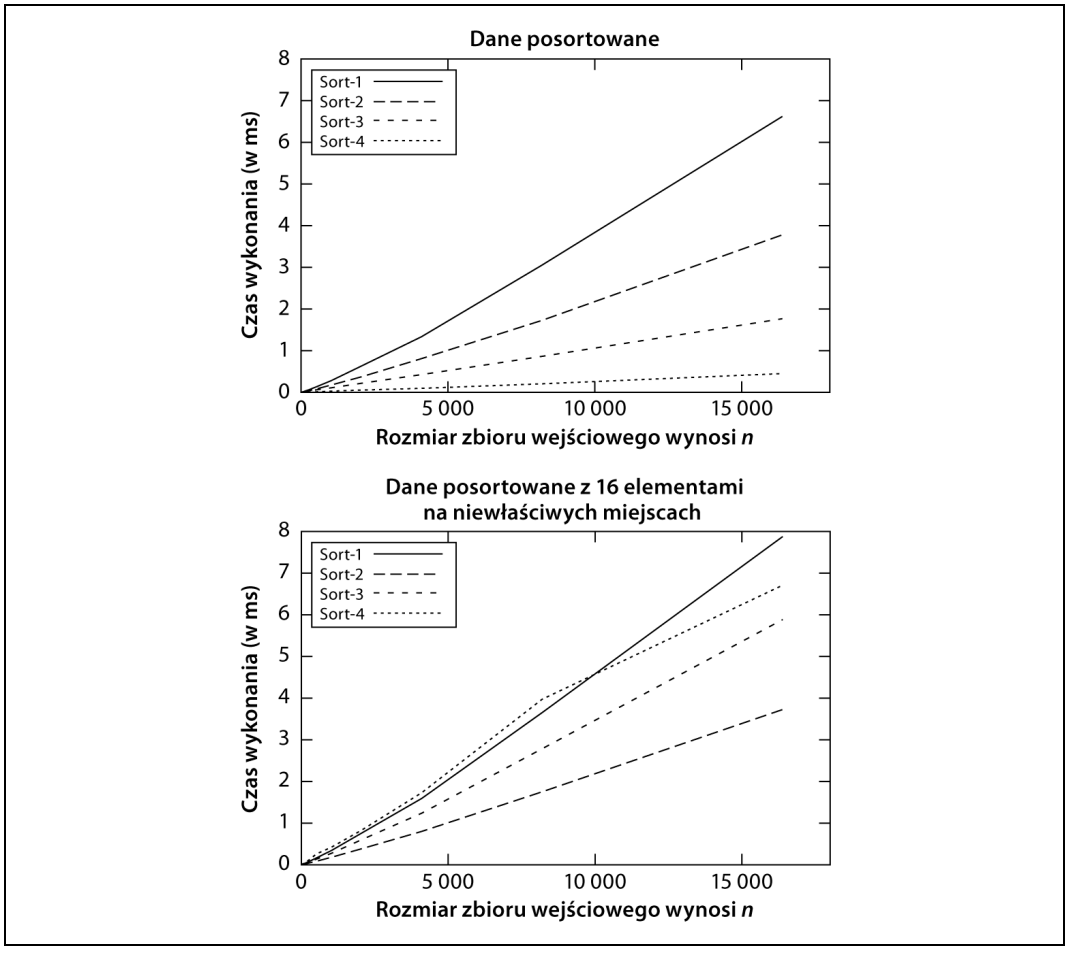
Rodzi się pytanie, czy wyniki z poprzedniego podrozdziału będą ważne dla wszystkich konkretyzacji problemu. A może SORT-2 jest najlepszy tylko do sortowania małej liczby napisów? Dane wejściowe mogą się przecież różnić pod wieloma względami:

- Napisów mogłoby być milion. Jaka będzie skalowalność algorytmu dla tak dużego wejścia?
- Dane mogą być częściowo posortowane, to znaczy, prawie wszystkie elementy nie są zbyt oddalone od swoich miejsc na posortowanym wykazie.

[6] Zob. np. *http://lxr.linux.no/linux+v2.6.11/fs/xfs/support/qsort.c*.

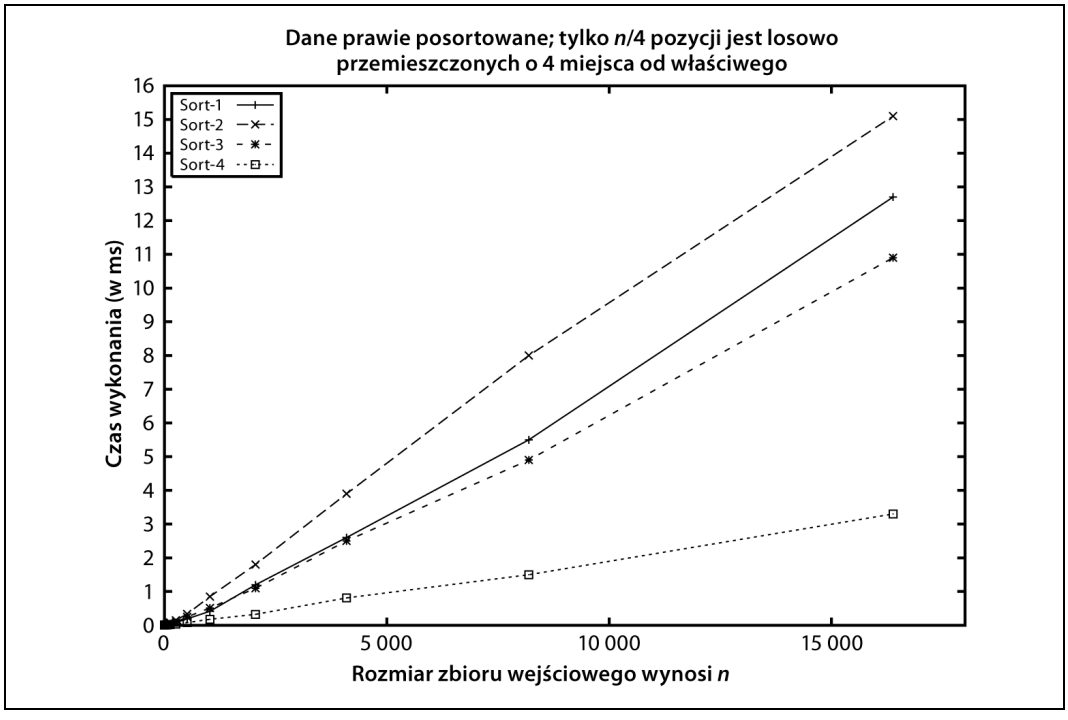
- Te same wartości mogą występować w danych wejściowych wielokrotnie.
- Niezależnie od rozmiaru *n* zbioru wejściowego elementy mogłyby pochodzić ze znacznie mniejszego zbioru i mogłoby się znajdować pośród nich wiele takich samych wartości.

Choć SORT-4 z rysunku 2.2 był najwolniejszy ze wszystkich czterech algorytmów w sortowaniu *n* losowych napisów, okazuje się, że jest on najszybszy, gdy dane są już posortowane. Ta przewaga jednak szybko zanika — wystarczy, aby losowych 16 elementów nie znajdowało się na właściwych miejscach (zob. rysunek 2.3).



Rysunek 2.3. Porównanie algorytmów sortowania przy sortowaniu danych już posortowanych lub prawie posortowanych

Przypuśćmy jednak, że tablica wejściowa z *n* napisami jest „prawie posortowana", tzn. $n/4$ napisów (25 procent) pozostaje na zamienionych pozycjach, lecz tylko w odległości czterech miejsc od właściwej. Można się zdziwić, widząc na rysunku 2.4, że SORT-4 jest wówczas lepszy od innych.



Rysunek 2.4. Algorytm Sort-4 wygrywa w przypadku danych prawie posortowanych

Wnioski, które można z tego wyciągnąć, są takie, że dla wielu problemów nie istnieje jeden optymalny algorytm. Wybór algorytmu zależy od zrozumienia rozwiązywanego problemu; wypada też uwzględnić rozkłady prawdopodobieństwa w konkretnych danych oraz zachowanie poszczególnych algorytmów.

Aby dostarczyć pewnych wskazówek, algorytmy są zazwyczaj przedstawiane z uwzględnieniem trzech typowych przypadków:

Przypadek najgorszy
: Definiuje klasę danych wejściowych, dla której algorytm wykazuje najgorszy czas działania. Zamiast określania konkretnych danych projektanci na ogół opisują *właściwości* danych wejściowych, które powodują, że algorytm nie działa wydajnie.

Przypadek średni
: Definiuje zachowanie oczekiwane w przypadku wykonywania algorytmu na losowych danych wejściowych. Mówiąc nieformalnie, nawet jeśli dla jakichś egzemplarzy problemu trzeba będzie zużyć więcej czasu z jakichś specjalnych powodów, zdecydowana większość problemów wejściowych nie będzie odbiegać od tej normy. Miara taka określa, na co może liczyć przeciętny użytkownik algorytmu.

Przypadek najlepszy
: Definiuje klasę danych wejściowych, na której algorytm działa najlepiej. Z takimi danymi algorytm ma najmniej do roboty. W rzeczywistości przypadki najlepsze zdarzają się rzadko.

Znając sprawność algorytmu w tych poszczególnych przypadkach, możesz rozstrzygnąć, czy jest on odpowiedni w Twojej konkretnej sytuacji.

Przypadek najgorszy

Wraz ze wzrostem n większość problemów ma więcej możliwych konkretyzacji rozmiaru n. Dla dowolnej konkretnej wartości n ilość pracy wykonana przez algorytm lub program może być zdecydowanie różna, zależnie od egzemplarza problemu rozmiaru n. Dla danego programu i wartości n czas działania w przypadku najgorszym jest maksymalnym czasem wykonania wybranym spośród wszystkich czasów działania na egzemplarzach rozmiaru n.

Przykładanie wagi do przypadku najgorszego jest pesymistycznym widzeniem świata. Najgorszy przypadek zachowania algorytmu interesuje nas z powodu:

potrzeby uzyskania odpowiedzi
: analiza złożoności algorytmu jest w tym przypadku często najłatwiejsza;

ograniczeń czasu rzeczywistego
: jeśli planujesz system do pomagania chirurgowi wykonującemu operację na otwartym sercu, to jest nie do przyjęcia, aby program działał nadspodziewanie długo[7] (nawet jeśli takie wolne działanie nie zdarza się „często").

Ujmując rzecz bardziej formalnie, jeśli S_n jest zbiorem konkretyzacji problemu s_i rozmiaru n, a t oznacza czas wykonania algorytmu w każdym z tych przypadków, to działanie algorytmu na S_n w przypadku najgorszym wynosi maksimum $t(s_i)$, gdzie $s_i \in S_n$. Jeśli oznaczyć najgorszy przypadek działania na S_n przez $T_{wc}(n)$, to tempo rośnięcia $T_{wc}(n)$ określa złożoność algorytmu w najgorszym przypadku.

Ogólnie biorąc, nie wystarczy zasobów, aby obliczyć algorytm dla każdego przypadku s_i po to, żeby określić doświadczalnie problem wejściowy prowadzący do przypadku najgorszego działania. Zamiast tego oponent próbuje wysmażyć najgorszy przypadek problemu wejściowego na podstawie opisu algorytmu.

Przypadek średni

System telefoniczny zaprojektowany do obsługi dużej liczby n telefonów musi w najgorszym przypadku zdołać obsłużyć wszystkie rozmowy, gdy $n/2$ osób chwyta za słuchawki i dzwoni do innych $n/2$ osób. Choć system taki nie ulegałby nigdy awarii z powodu przeciążenia, cena jego budowy byłaby zaporowa. Poza tym prawdopodobieństwo, że każda z $n/2$ osób dzwoni do innej spośród pozostałych $n/2$ osób, jest nadzwyczaj małe. Można zaprojektować system tańszy, a mimo to bardzo rzadko doznający załamań z powodu przeciążenia (możliwe, że nigdy). Musimy się jednak odwołać do aparatu matematycznego, aby rozważyć prawdopodobieństwa.

Ze zbiorem konkretnych problemów rozmiaru n kojarzymy rozkład prawdopodobieństwa Pr, w którym prawdopodobieństwa z przedziału od 0 do 1 są przypisane każdemu problemowi w ten sposób, że suma prawdopodobieństw wszystkich konkretnych problemów rozmiaru n wynosi 1. Nieco bardziej formalnie, jeśli S_n jest zbiorem konkretyzacji rozmiaru n, to

$$\sum_{s_{i \in S_n}} \Pr\{s_i\} = 1$$

[7] W systemach czasu rzeczywistego mniej chodzi o szybkość działania, a bardziej o terminową i niezawodną obsługę w ustalonym z góry czasie; to jest istotna różnica — *przyp. tłum.*

Jeśli t jest miarą pracy wykonanej przez algorytm w każdym z przypadków, to praca wykonana przez algorytm na S_n w przypadku średnim jest określona następująco:

$$T_{\text{ac}}(n) = \frac{1}{|S_n|} \sum_{s_i \in S_n} t(s_i) \Pr\{s_i\}$$

Oznacza to, że faktyczna praca $t(s_i)$ w przypadku s_i jest ważona prawdopodobieństwem tego, że s_i wystąpi jako dane wejściowe. Jeśli $\Pr\{s_i\} = 0$, to faktyczna wartość $t(s_i)$ nie wpływa na oczekiwaną pracę wykonywaną przez program. Jeśli oznaczyć przypadek średni na S_n przez $T_{\text{ac}}(n)$, to tempo rośnięcia $T_{\text{ac}}(n)$ określa złożoność algorytmu w przypadku średnim.

Przypomnijmy, że opisując tempo rośnięcia pracochłonności lub czasu, konsekwentnie pomijamy stałe. Jeśli więc mówimy, że WYSZUKIWANIE LINIOWE ze zbioru n elementów zajmuje średnio

$$\frac{1}{2}n + \frac{1}{2}$$

badań (stosownie do naszych wcześniejszych założeń), to na zasadzie umowy mówimy po prostu, że wedle tych przypuszczeń oczekujemy, iż WYSZUKIWANIE LINIOWE będzie sprawdzać *liniową* liczbę elementów[8], czyli *rzędu n*.

Przypadek najlepszy

Znajomość najlepszego przypadku działania algorytmu jest przydatna, mimo że sytuacja taka rzadko występuje w praktyce. W wielu wypadkach daje ona wgląd w optymalne warunki danego algorytmu. Na przykład, najlepszy przypadek WYSZUKIWANIA LINIOWEGO występuje wówczas, gdy poszukuje się wartości v, która znajduje się na początku wykazu. Nieco odmienna metoda, którą będziemy nazywać WYSZUKIWANIEM ZE ZLICZANIEM, polega na poszukiwaniu danej wartości v i zliczaniu, ile razy wystąpiła ona na wykazie. Jeśli licznik po obliczeniu wynosi 0, znaczy to, że elementu nie znaleziono, toteż algorytm zwraca `false`; w przeciwnym razie zwraca `true`. Zauważmy, że WYSZUKIWANIE ZE ZLICZANIEM zawsze dociera do końca wykazu, więc choć jego najgorszy przypadek ma złożoność O(n) — taką samą jak WYSZUKIWANIE LINIOWE — zachowanie algorytmu w przypadku najlepszym również jest rzędu O(n); do poprawy jego działania nie można więc wykorzystać ani przypadku najlepszego, ani średniego.

Rodziny efektywności

Nasze porównywanie algorytmów opieramy na ocenie ich sprawności dla danych wejściowych rozmiaru n. Jest to metodyka standardowa, opracowana do porównywania algorytmów ponad pół wieku temu. Postępując w ten sposób, możemy określić, które algorytmy skalują się dobrze na problemy niebanalnych rozmiarów, obliczając czas zużywany przez algorytm w powiązaniu z rozmiarem dostarczonego wejścia. Dodatkową postacią oceny efektywności jest uwzględnienie, ile pamięci algorytm potrzebuje do działania. Tymi zagadnieniami zajmujemy się, stosownie do potrzeb, w poszczególnych rozdziałach z algorytmami.

[8] Będzie je sprawdzać w czasie liniowym, rosnącym liniowo — *przyp. tłum.*

W książce stosujemy wyłącznie następujące kategorie, uporządkowane tutaj według malejącej sprawności:

- stała,
- logarytmiczna,
- podliniowa,
- liniowa,
- n log (n),
- kwadratowa,
- wykładnicza.

Przedstawimy teraz w kilku punktach niektóre z tych kategorii sprawności.

Omówienie 0. Zachowanie stałe

Analizując działanie algorytmów w tej książce, często przyjmujemy, że sprawność pewnych elementarnych operacji jest stała. Nie przesądza to oczywiście o faktycznym działaniu operacji, gdyż nie odnosimy się do żadnego konkretnego sprzętu. Na przykład, porównanie, czy dwie 32-bitowe liczby x i y są takie same, powinno być wykonywane tak samo sprawnie niezależnie od wartości x i y. Operację działającą w czasie stałym określa się jako mającą sprawność O(1).

A co ze sprawnością porównywania liczb 256-bitowych? Albo dwóch liczb 1024-bitowych? Otóż dla ustalonego z góry rozmiaru k porównania dwóch liczb k-bitowych możesz dokonać w stałym czasie. Obowiązuje tu zasada, że rozmiar problemu (tj. wartości porównywanych liczb x i y) nie może przekroczyć ustalonej wartości k. Dodatkowy nakład, będący jako k mnożnikiem stałym, zaniedbujemy w notacji O(1).

Omówienie 1. Zachowanie log n

Barman oferuje każdemu chętnemu zakład o 10 000 dolarów: „Wybiorę liczbę z przedziału od 1 do 1 000 000, a ty spróbuj ją odgadnąć, typując kolejno (do) 20 liczb; po każdej próbie odgadnięcia powiem ci: ZA DUŻA, ZA MAŁA lub TRAFIONA. Jeśli wygrasz w 20 pytaniach, dam ci 10 000 dolarów, w przeciwnym razie ty dasz mi 10 000 dolarów". Przyjmujesz ten zakład? Warto, ponieważ zawsze możesz go wygrać. W tabeli 2.1 pokazano przykładowy przebieg zdarzeń dla przedziału od 1 do 10, w którym każde z serii zadanych pytań zmniejsza rozmiar problemu o połowę.

W każdej kolejce, zależnie od odpowiedzi udzielonej przez barmana, rozmiar potencjalnego przedziału z ukrytą liczbą jest obcinany o około połowę. Na koniec przedział z ukrytą liczbą zostaje ograniczony do jednej liczby; dochodzi do tego po $\lceil \log(n) \rceil$ próbach. Funkcja sufitowa $\lceil x \rceil$ zaokrągla liczbę x do najmniejszej liczby całkowitej większej lub równej x. Jeśli barman wybierze na przykład liczbę między 1 a 10, to zdołasz ją zgadnąć w $\lceil \log(10) \rceil = \lceil 3{,}32 \rceil$, czyli w 4 próbach, jak pokazano w tabeli.

Działa to tak samo dla miliona liczb. Algorytm ZGADYWANIE, pokazany w przykładzie 2.1, działa w istocie dla dowolnego przedziału [*dolne, górne*] i wyznacza wartość n po $\lceil \log(górne - dolne + 1) \rceil$ krokach. Jeśli liczb jest milion, to algorytm znajdzie liczbę w co najwyżej $\lceil \log(1\,000\,000) \rceil = 19{,}93$, czyli 20 krokach (przypadek najgorszy).

Tabela 2.1. Przykład postępowania przy zgadywaniu liczb od 1 do 10

Liczba	Pierwsze zgadywanie	Drugie zgadywanie	Trzecie zgadywanie
1	Czy to jest 5? ZA DUŻA	Czy to jest 2? ZA DUŻA	Czy to jest 1? TRAFIONA
2	Czy to jest 5? ZA DUŻA	Czy to jest 2? TRAFIONA	
3	Czy to jest 5? ZA DUŻA	Czy to jest 2? ZA MAŁA	Czy to jest 3? TRAFIONA
4	Czy to jest 5? ZA DUŻA	Czy to jest 2? ZA MAŁA	Czy to jest 3? ZA MAŁA, więc musi to być 4
5	Czy to jest 5? TRAFIONA		
6	Czy to jest 5? ZA MAŁA	Czy to jest 8? ZA DUŻA	Czy to jest 6? TRAFIONA
7	Czy to jest 5? ZA MAŁA	Czy to jest 8? ZA DUŻA	Czy to jest 6? ZA MAŁA, więc musi to być 7
8	Czy to jest 5? ZA MAŁA	Czy to jest 8? TRAFIONA	
9	Czy to jest 5? ZA MAŁA	Czy to jest 8? ZA MAŁA	Czy to jest 9? TRAFIONA
10	Czy to jest 5? ZA MAŁA	Czy to jest 8? ZA MAŁA	Czy to jest 9? ZA MAŁA, więc musi to być 10

Przykład 2.1. Kod w Javie do zgadywania liczby w przedziale [dolne, górne]

```
// Wyznacza liczbę kroków, gdy n na pewno znajduje się w przedziale
// [dolne, górne] ([low, high]).
public static int turns(int n, int low, int high) {
   int turns = 0;

   // Wykonuj, dopóki do sprawdzenia pozostają więcej niż 2 liczby.
   while (high – low <= 2) {
      // Przygotuj punkt środkowy [low, high] jako wybór.
      turns++;
      int mid = (low + high)/2;
      in (mid == n) {
         return turns;
      } else if (mid < n) {
         low = mid + 1;
      } else {
         high = mid – 1;
      }
   }
   // Tutaj zostały już tylko dwie liczby. Wybieramy jedną z nich i jeśli to
   // nie ta, to odgadywaną jest druga. Dochodzi więc tylko jeden krok.
   return 1 + turns;
}
```

Algorytmy *logarytmiczne* są wyjątkowo sprawne, ponieważ szybko dążą do rozwiązania. Swoją popularność zawdzięczają temu, że w każdym kroku zmniejszają rozmiar problemu mniej więcej o połowę. Algorytm ZGADYWANIE dociera do rozwiązania po co najwyżej $k = \lceil \log(n) \rceil$ iteracjach, a w i-tym powtórzeniu ($i > 0$) algorytm zgaduje odpowiedź, o której wiadomo, że należy do przedziału $\pm\varepsilon = 2^{k-i}$ wokół poszukiwanej liczby. Wielkość ε jest określana mianem błędu lub niepewności. Po każdym powtórzeniu pętli ε jest obcinane o połowę.

Innym przykładem ukazującym wydajne działanie jest metoda Newtona obliczania pierwiastków równań z jedną zmienną (innymi słowy, dla jakich wartości x $f(x) = 0$). Aby znaleźć, kiedy $x \cdot sin(x) - 5 \cdot x = cos(x)$, weźmy $f(x) = x \cdot sin(x) - 5 \cdot x - cos(x)$ i jej *pochodną* $f'(x) = x \cdot cos(x) + sin(x) - 5 - sin(x) = x \cdot cos(x) - 5$. W kolejnym kroku metody Newtona jest obliczane $x_{n+1} = x_n - f(x_n)/f'(x_n)$. Zaczynając od „odgadniętej" wartości x, którą jest 0, algorytm ten szybko wyznacza poprawne rozwiązanie $x = -0{,}189302759$, co pokazano w tabeli 2.2. Cyfry dwójkowe i dziesiętne w nawiasach [] oznaczają cyfry dokładne.

Tabela 2.2. Metoda Newtona

n	x_n dziesiętnie	x_n w bitach (cyfrach dwójkowych)
0	0.0	
1	–0,2	[101111111001]0011001100110011001100110...
2	–[0,18]8516717588...	[101111111001000001]0000101010000110110...
3	–[0,1893]59749489...	[101111111001000001111]10011110000101101...
4	–[0,189]298621848...	[101111111001000001111011101111111011...
5	–[0,18930]3058226...	[101111111001000001111011000101010010001...
6	–[0,1893027]36274...	[101111111001000001111011000100110011100...
7	–[0,189302759]639...	[101111111001000001111011000100101]01001...

Omówienie 2. Zachowanie podliniowe O(n^d) dla d<1

W pewnych sytuacjach działanie algorytmu jest lepsze niż *liniowe*, lecz nie tak wydajne jak *logarytmiczne*. Jak omówiono w rozdziale 9., kd-drzewo w wielu wymiarach może sprawnie podzielić zbiór n d-wymiarowych punktów. Jeśli drzewo jest zrównoważone, to czas wyszukiwania dotyczącego zapytań przedziałowych, zgodnych z osiami punktów, wynosi $O(n^{1-1/d})$.

Omówienie 3. Sprawność liniowa

Rozwiązywanie niektórych problemów wymaga wyraźnie większych nakładów pracy niż innych. Każdy ośmiolatek obliczy, że 7+5 to 12. O ile trudniejszy jest problem 37+45?

Mówiąc ogólnie, ile zachodu wymaga dodanie dwóch n-cyfrowych liczb $a_n \ldots a_1 + b_n \ldots b_1$, aby otrzymać $c_{n+1} \ldots c_1$ cyfr wyniku? Oto elementarne operacje używane w algorytmie DODAWANIE:

$$c_i \leftarrow (a_i + b_i + przeniesienie_i) \bmod 10$$

$$przeniesienie_{i+1} \leftarrow \begin{cases} 1, \text{ gdy } a_i + b_i + przeniesienie_i \geq 10 \\ 0 \text{ w przeciwnym razie} \end{cases}$$

Prostą realizację DODAWANIA w Javie przedstawiono jako przykład 2.2; liczba n-cyfrowa jest w niej reprezentowana jak tablica wartości `int`. W przykładach w tym podrozdziale przyjęto, że każda z tych wartości jest liczbą d reprezentującą cyfrę dziesiętną, taką że $0 \leq d \leq 9$.

Przykład 2.2. Realizacja dodawania (add) w Javie

```
public static void add(int[] n1, int[] n2, int[] sum) {
   int position = n1.length-1;
   int carry = 0;
```

```
    while (position >= 0) {
       int total = n1[position] + n2[position] + carry;
       sum[position+1] = total % 10;
       if (total > 9) { carry = 1; } else { carry = 0; }
       position--;
    }
    sum[0] = carry;
}
```

Jeśli tylko problem wejściowy mieści się w pamięci, metoda `add` oblicza sumę dwóch liczb reprezentowanych przez tablice `n1` i `n2`. Czy ta realizacja jest równie sprawna jak inna, nazwana `last`, i pokazana jako przykład 2.3?

Przykład 2.3. Realizacja last w Javie

```
public static void last(int[] n1, int[] n2, int[] sum) {
   int position = n1.length;
   int carry = 0;
   while (--position >= 0) {
      int total = n1[position] + n2[position] + carry;
      if (total > 9) {
         sum[position+1] = total - 10;
         carry = 1;
      } else {
         sum[position+1] = total;
         carry = 0;
      }
   }
   sum[0] = carry;
}
```

Czy te na pozór małe różnice realizacyjne wpływają na sprawność algorytmu? Weźmy pod uwagę dwa inne czynniki, które potencjalnie mogą wpływać na sprawność algorytmu:

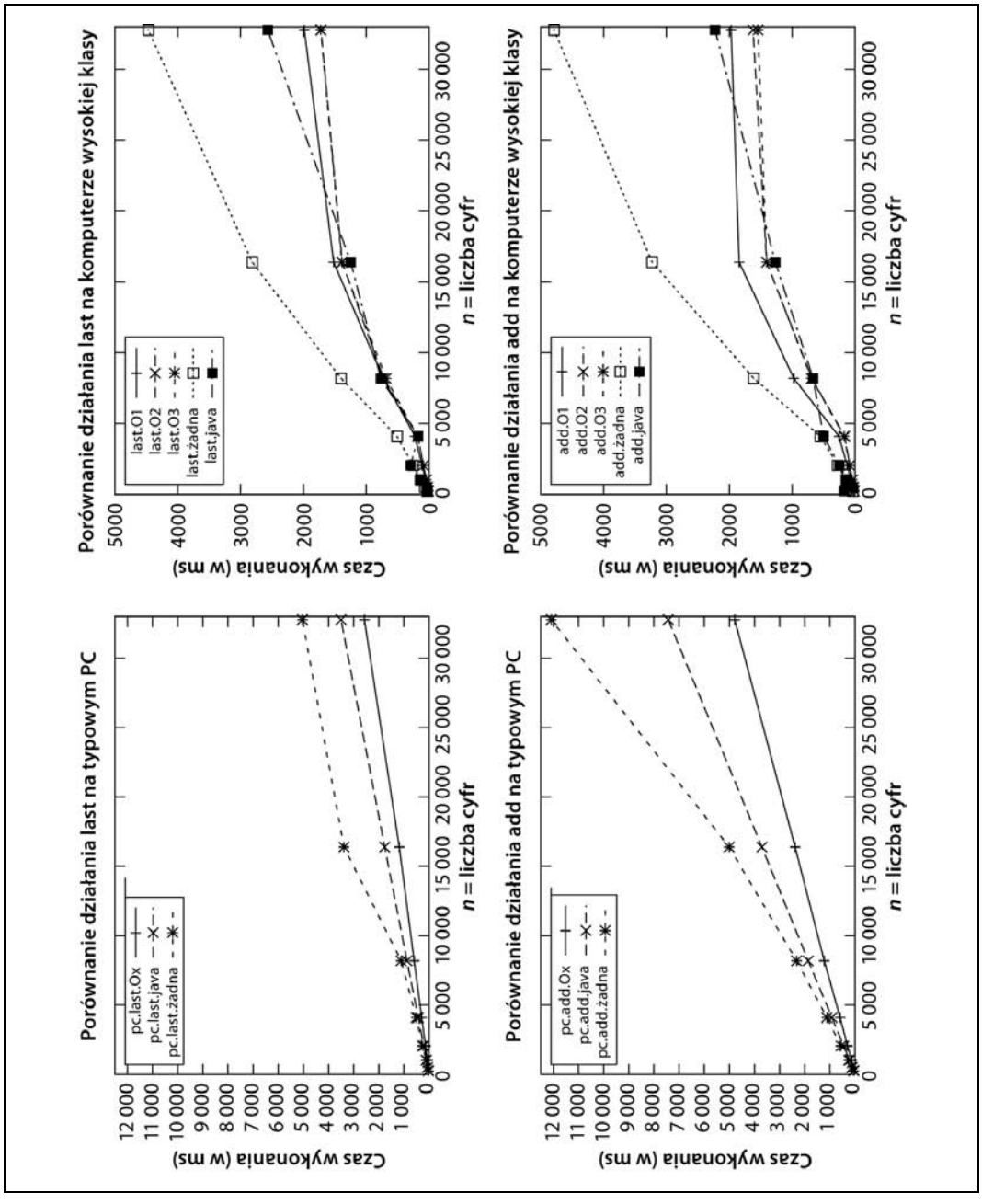
- Jednym czynnikiem jest język programowania. Metody `add` i `last` można bez trudu zamienić na programy w języku C. Jak wybór języka wpływa na sprawność algorytmu?
- Programy mogą być wykonywane na różnych komputerach. Jak wybór sprzętu komputerowego wpływa na sprawność algorytmu?

Realizacje te były wykonywane po 10 000 razy na liczbach o długości od 256 do 32 768 cyfr. Dla każdego rozmiaru generowano losowo liczbę tego rozmiaru; dalej — w każdej z tych 10 000 prób liczba ta była przesuwana cyklicznie (raz w prawo, raz w lewo), aby powstały dwie różne liczby do dodania. Użyto następujących komputerów: typowego PC-ta i komputera wysokiej klasy, jak omówiono w rozdziale 10. Posłużono się dwoma językami programowania (C i Java). Zaczęliśmy od hipotezy, że wraz z podwojeniem problemu czas wykonania algorytmu ulegnie podwojeniu. Chcieliśmy się również upewnić, że zachowanie to — z grubsza biorąc — występuje niezależnie od użytej maszyny, języka programowania czy implementacji.

Na rysunku 2.5 przedstawiono wykres czasu wykonania 10 000 obliczeń (czas ten jest reprezentowany na osi Y) w funkcji rozmiaru problemu (przypisano mu oś X). Każdy wariant został wykonany na zbiorze konfiguracji:

g
: wersja w języku C skompilowana z dołączonymi informacjami uruchomieniowymi;

żadna
: wersja w języku C skompilowana bez żadnych optymalizacji;



Rysunek 2.5. Porównanie realizacji funkcji add i last w różnych scenariuszach

O1, O2, O3
: wersja w języku C skompilowana na trzech różnych poziomach optymalizacji; ogólnie, im wyższy numer poziomu, tym lepsza optymalizacja, a co za tym idzie — spodziewana sprawność;

Java
: wersje algorytmów w Javie;

PC-Java
: to jedyna konfiguracja wykonana na PC; poprzednie były wykonywane na komputerze wysokiej klasy.

Zauważmy, że każdą z linii obliczonych na wykresach po lewej stronie rysunku 2.5 (oznaczonych „typowy PC") można przybliżyć funkcją liniową, co podtrzymuje założenie, że między wartościami X i Y istnieje zależność liniowa. Obliczeń wykonanych na komputerze wysokiej klasy nie da się tak łatwo zakwalifikować jako liniowych, co sugeruje, że wpływ procesora wyższej klasy jest istotny.

W tabeli 2.3 przedstawiono podzbiór wykreślonych danych w postaci numerycznej. Stosowne informacje generuje kod uzupełniający książkę. W ostatniej, siódmej kolumnie porównano bezpośrednio czasy działania realizacji HighEnd-C-Last-O3[9], wykazane w szóstej kolumnie. Proporcja czasów działania wynosi prawie 2, jak oczekiwano. Niech $t(n)$ będzie rzeczywistym czasem działania algorytmu DODAWANIE na danych rozmiaru n. Krzywa tego wzrostu dostarcza empirycznego dowodu, że czas w milisekundach potrzebny do obliczenia funkcji `last` dla dwóch liczb n-cyfrowych na komputerze wysokiej klasy z użyciem realizacji w C i z optymalizacją na poziomie –O3 będzie się mieścić między $n/11$ a $n/29$.

Tabela 2.3. Czas (w milisekundach) wykonania 10 000 wywołań add lub last na liczbach losowych rozmiaru n

n	PC-Java-add	Wysokiej klasy, Java-add	Wysokiej klasy, C-add-żadna	Wysokiej klasy, C-add-O3	Wysokiej klasy, C-last-O3	Proporcja ostatniej kolumny i rozmiaru
256	60	174	34	11	9	
512	110	36	70	22	22	2,44
1024	220	124	139	43	43	1,95
2048	450	250	275	87	88	2,05
4096	921	500	550	174	180	2,05
8192	1861	667	1611	696	688	3,82
16 384	3704	1268	3230	1411	1390	2,02
32 768	7430	2227	4790	1555	1722	1,24
65 536	17 453	2902	9798	3101	3508	2,04
131 072	35 860	12 870	20 302	7173	7899	2,25
262 144	68 531	22 768	41 800	14 787	16 479	2,09
524 288	175 015	31 148	82 454	29 012	32 876	2
1 048 576	505 531	64 192	162 955	53 173	63 569	1,93

Informatycy teoretycy sklasyfikowaliby algorytm DODAWANIE jako *liniowy* względem rozmiaru jego wejścia n. To znaczy, istnieje pewna stała $c > 0$, taka że $t(n) \leq c \cdot n$ dla każdego $n > n_0$. Nie musimy w istocie znać dokładnie wartości c ani n_0 — wystarczy, że istnieją. Można udowodnić, że jest możliwe ustalenie dolnego ograniczenia czasu liniowego złożoności dodawania przez wykazanie, że należy sprawdzić każdą cyfrę (rozważ skutki niezbadania którejś z cyfr).

[9] Czyli realizacji funkcji `last` w języku C, skompilowanej z parametrem optymalizacji na poziomie –O3 i wykonanej na komputerze wysokiej klasy, jak opisano w dodatku zawierającym testy wzorcowe.

W realizacji `last` algorytmu DODAWANIE możemy ustawić c na 1/11 i wybrać 256 jako n_0. Inne realizacje DODAWANIA będą miały różne stałe, niemniej ich ogólne zachowanie pozostanie *liniowe*. Wynik ten może się wydawać dziwny, zważywszy, że większość programistów zakłada, iż działania całkowite są wykonywane w stałym czasie; stały czas dodawania osiąga się jednak tylko wówczas, gdy reprezentacja liczby całkowitej (np. 16- lub 64-bitowa) ma stały rozmiar całkowity n[10].

W porównywaniu algorytmów stała c nie odgrywa takiej roli jak znajomość rzędu algorytmu. Na pozór drobne różnice spowodowały różne działanie. Realizacja `last` algorytmu DODAWANIE jest wyraźnie wydajniejsza po wyeliminowaniu operatora modulo (`%`), który jest znany z powolności, jeśli działa na wartościach nie będących potęgami 2. W rozpatrywanym przykładzie niewydajne jest właśnie działanie „`% 10`”, gdyż musi tu wystąpić dzielenie przez 10, które na komputerach binarnych jest operacją kosztowną. Nie oznacza to, że ignorujemy wartość c. Jest oczywiste, że jeśli wykonujemy DODAWANIE wielką liczbę razy, to nawet małe zmiany wartości c mogą istotnie oddziałać na sprawność programu.

Omówienie 4. Sprawność n log n

Typowe zachowanie wydajnych algorytmów najlepiej charakteryzuje ta rodzina sprawności. Aby wyjaśnić, jak to zachowanie objawia się w praktyce, określmy $t(n)$ jako czas zużywany przez algorytm na rozwiązanie problemu o danych wejściowych rozmiaru n. Skutecznym sposobem rozwiązywania problemów jest metoda „dziel i zwyciężaj”, w której problem rozmiaru n jest dzielony na (z grubsza równej wielkości) podproblemy rozmiaru $n/2$, które są rozwiązywane rekurencyjnie, a ich rozwiązania są łączone w pewien sposób, aby uzyskać rozwiązanie problemu oryginalnego, rozmiaru n. Matematycznie można to wyrazić tak:

$$t(n) = 2 \cdot t(n/2) + \mathrm{O}(n)$$

Oznacza to, że $t(n)$ zawiera koszt dwóch podproblemów oraz nie więcej niż koszt liniowego czasu połączenia wyników. Po prawej stronie równania widzimy $t(n/2)$, czyli czas rozwiązania problemu rozmiaru $n/2$; na tej samej zasadzie możemy zapisać, że

$$t(n/2) = 2 \cdot t(n/4) + \mathrm{O}(n/2)$$

zatem pierwotne równanie przyjmie teraz postać:

$$t(n) = 2 \cdot [2 \cdot t(n/4) + \mathrm{O}(n/2)] + \mathrm{O}(n)$$

Jeśli postąpimy tak po raz kolejny, otrzymamy:

$$t(n) = 2 \cdot [2 \cdot [2 \cdot t(n/8) + \mathrm{O}(n/4)] + \mathrm{O}(n/2)] + \mathrm{O}(n)$$

Ostatnie równanie redukuje się do $t(n) = 8 \cdot t(n/8) + \mathrm{O}(3 \cdot n)$. Uogólniając, możemy powiedzieć, że $t(n) = 2^k \cdot t(n/2^k) + \mathrm{O}(k \cdot n)$. Rozwinięcie to kończy się, gdy $2^k = n$, czyli gdy $k = \log(n)$. W ostatecznym podstawowym przypadku, gdy problem ma rozmiar 1, sprawność $t(1)$ jest stałą c. Widzimy więc, że zamknięty wzór ma postać $t(n) = n \cdot c + \mathrm{O}(n \cdot \log(n))$. Ponieważ $n \cdot \log(n)$ jest asymptotycznie większe niż $c \cdot n$ dla dowolnie wybranej stałej c, $t(n)$ można prościej zapisać jako $\mathrm{O}(n \log n)$.

[10] Musiałby to być naprawdę gapowaty programista, żeby nie zauważyć, iż omawiane funkcje `add` i `last` są w istocie działaniami na wektorach (maszynowych) liczb całkowitych — *przyp. tłum.*

Omówienie 5a. Sprawność kwadratowa

Rozważmy teraz podobny problem, w którym dwie liczby całkowite rozmiaru *n* są mnożone przez siebie. Przykład 2.4 ukazuje realizację MNOŻENIA, elementarnego algorytmu znanego ze szkoły.

Przykład 2.4. Realizacja mnożenia (mult) w Javie

```
public static void mult(int[] n1, int[] n2, int[] result) {
   int pos = result.length-1;

   // Wyczyść wszystkie wartości...
   for (int i = 0; i < result.length; i++) { result[i] = 0; }
   for (int m = n1.length-1; m >= 0; m--) {
      int off = n1.length-1 - m;
      for (int n = n2.length-1; n >= 0; n--, off++) {
         int prod = n1[m]*n2[n];
         // Oblicz sumę częściową, przenosząc z poprzedniej pozycji
         result[pos-off] += prod % 10;
         result[pos-off-1] += result[pos-off]/10 + prod/10;
         result[pos-off] %= 10;
      }
   }
}
```

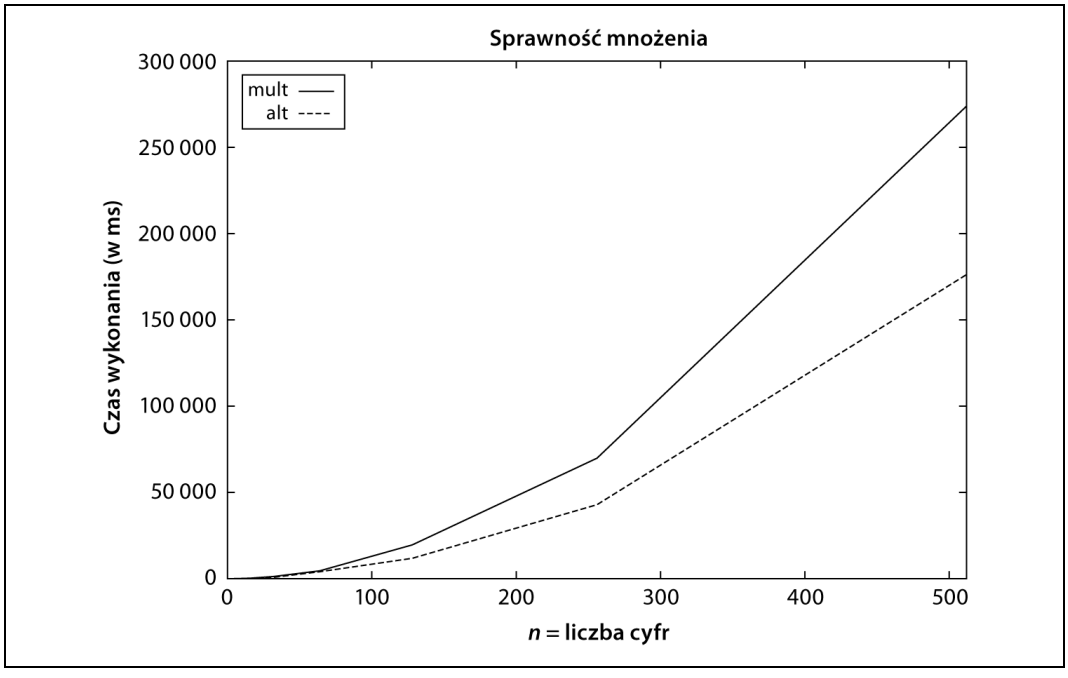
Podobnie jak poprzednio, napisano alternatywny program `alt`, który eliminuje konieczność używania kosztownego operatora modulo i przeskakuje najbardziej wewnętrzne obliczenia, gdy `n1[m]` wynosi 0 (`alt` nie jest tu pokazany, lecz można go znaleźć w udostępnionym magazynie kodu). Wersja `alt` ma 203 wiersze kodu wygenerowanego w Javie, żeby usunąć dwa operatory modulo. Czy ta odmiana zaoszczędza kosztów, co usprawiedliwiłoby dodatkowe nakłady na dopracowanie i wypielęgnowanie tego wygenerowanego kodu?

W tabeli 2.4 pokazano zachowanie obu realizacji algorytmu MNOŻENIE z użyciem tych samych losowych danych wejściowych, których użyto do zademonstrowania działania DODAWANIA. Działanie algorytmu przedstawiono w postaci graficznej na rysunku 2.6; uwidoczniono na nim paraboliczną krzywą wzrostu, która charakteryzuje zachowanie *kwadratowe*.

Tabela 2.4. Czas (w milisekundach) wykonania 10 000 mnożeń

n	$mult_n$(ms)	alt_n(ms)	$mult_{2n}$ / $mult_n$
2	15	0	
4	15	15	1
8	62	15	4,13
16	297	218	4,80
32	1187	734	4,00
64	4516	3953	3,80
128	19 530	11 765	4,32
256	69 828	42 844	3,58
512	273 874	176 203	3,92

Mimo że odmiana `alt` jest około 40% szybsza, zarówno `alt`, jak i `mult` wykazują tę samą sprawność asymptotyczną. Iloraz $mult_{2n}/mult_n$ wynosi około 4, co wskazuje, że sprawność MNOŻENIA jest *kwadratowa*. Niech $t(n)$ oznacza rzeczywisty czas algorytmu MNOŻENIE na



Rysunek 2.6. Porównanie mult i alt

danych rozmiaru n. Zgodnie z tą definicją musi istnieć stała $c > 0$, taka że $t(n) \leq c \cdot n^2$ dla każdego $n > n_0$. Nie musimy znać dokładnie wartości c i n_0 — wystarczy, że istnieją. Dla realizacji `mult` MNOŻENIA na naszej platformie możemy ustalić c jako 1,2 i za n_0 wybrać 64.

I znów okazuje się, że indywidualne zmiany w realizacji nie pomogły „przełamać" zachowania kwadratowego, tkwiącego w samym algorytmie. Istnieją jednak inne algorytmy [Zuras, 1994] mnożenia par liczb n-cyfrowych, znacznie szybsze niż kwadratowe. Algorytmy te są ważne w aplikacjach w rodzaju szyfrowania, w których często mnoży się duże liczby.

Omówienie 5b. Mniej oczywiste obliczenia sprawności

Najczęściej już samo przeczytanie opisu algorytmu wystarcza (jak pokazano w DODAWANIU i MNOŻENIU) do sklasyfikowania go jako *liniowy* lub *kwadratowy*. Podstawową wskazówką *kwadratowości* jest na przykład występowanie pętli zagnieżdżonej w innej pętli. Niektóre algorytmy nie poddają się jednak tak prostej analizie. Rozważmy algorytm GCD uwidoczniony w przykładzie 2.5, wymyślony przez Euklidesa i służący do obliczania największego wspólnego dzielnika (NWD) dwóch liczb całkowitych, których cyfry zapamiętano w tablicach.

Przykład 2.5. Algorytm Euklidesa

```
public static void gcd (int a[], int b[], int gcd[]) {
   if (isZero(a)) { assign (gcd, a); return; }
   if (isZero(b)) { assign (gcd, b); return; }

   // Zapewnij, że a i b nie zostaną zmienione
   a = copy (a);
   b = copy (b);
```

```
    while(!isZero(b)) {
       // Ostatni argument do odjęcia reprezentuje znak wyniku, który
       // możemy pominąć, gdyż odejmujemy tylko mniejsze od większych
       if (compareTo(a, b) > 0) {
          subtract (a, b, gcd, new int[1]);
          assign (a, gcd);
       } else {
          subtract (b, a, gcd, new int[1]);
          assign (b, gcd);
       }
    }

    // Wartość przechowywana w a jest obliczonym GCD liczb (a, b)
    assign (gcd, a);
 }
```

Algorytm powtarza porównywanie dwóch liczb (*a* i *b*) i odejmuje liczbę mniejszą od większej, aż dojdzie do zera. Realizacje pomocniczych metod (`isZero`, `assign`, `compareTo`, `substract`) nie są tu załączone, lecz można je znaleźć w uzupełniającym książkę magazynie kodu.

Algorytm ten wyznacza największy wspólny dzielnik (GCD, ang. *greatest common divisor*) dwóch liczb, lecz na podstawie rozmiaru danych wejściowych nie bardzo wiadomo, ile powtórzeń trzeba będzie wykonać. W każdym kroku pętli jest zmniejszane *a* lub *b* i żadne z nich nigdy nie staje się ujemne, co daje gwarancję, że algorytm się skończy, lecz obliczenie niektórych GCD trwa dłużej niż innych; na przykład wykonanie `gcd(1000,1)` ma 999 kroków! Wyraźnie widać, że działanie tego algorytmu jest bardziej uzależnione od wejścia niż DODAWANIA lub MNOŻENIA, ponieważ różne dane wejściowe tego samego rozmiaru wymagają bardzo różnych czasów obliczeń. Algorytm GCD wykazuje najgorsze zachowanie, gdy trzeba obliczyć GCD z $(10^n–1, 1)$; musi wówczas powtarzać pętlę `while` $(10^n–1)$ razy! Skoro wykazaliśmy już, że dodawanie i odejmowanie są rzędu $O(n)$ względem rozmiaru wejścia n, to GCD wymaga $n·(10^n–1)$ operacji w pętli. Zamieniając w tym równaniu podstawę na 2, otrzymujemy $n·(2^{3,3219·n})–n$, co oznacza sprawność wykładniczą. Klasyfikujemy ten algorytm jako $O(n·2^n)$.

Realizację `gcd` w przykładzie 2.5 z łatwością prześcignie algorytm MODGCD, przedstawiony w przykładzie 2.6, w którym posłużono się operatorem modulo do obliczenia całkowitej reszty z dzielenia *a* i *b*.

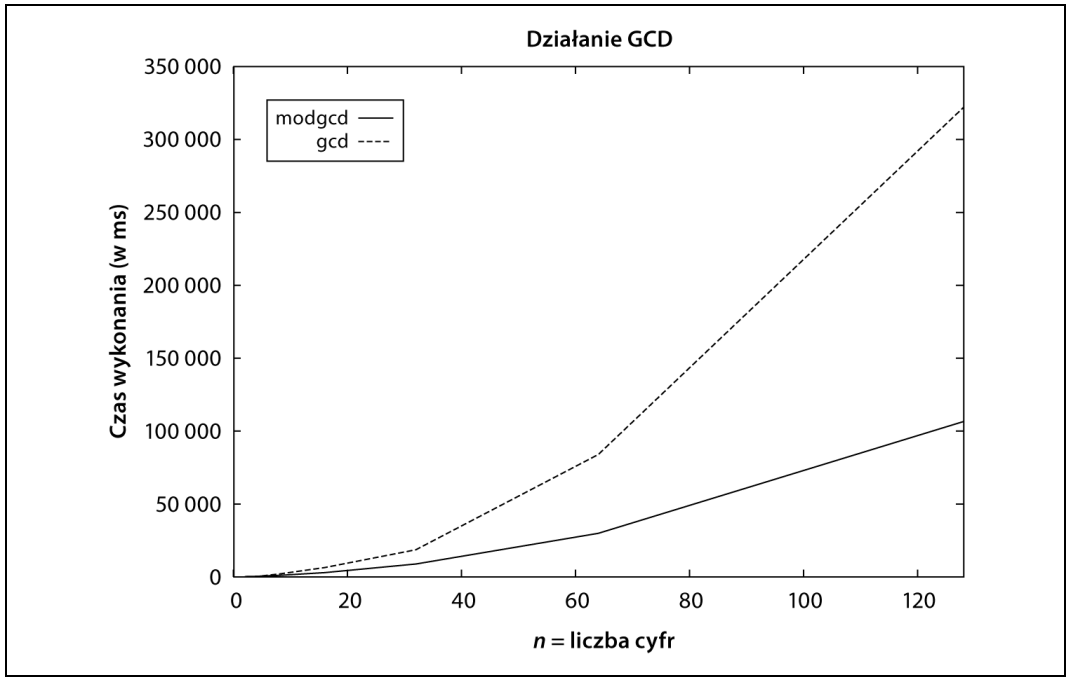
Przykład 2.6. Algorytm MODGCD obliczania GCD

```
public static void modgcd (int a[], int b[], int gcd[]) {
   if (isZero(a)) { assign (gcd, a); return; }
   if (isZero(b)) { assign (gcd, b); return; }

   // Wyrównaj liczbę cyfr w a i b, po czym pracuj na kopiach
   a = copy (normalize(a, b.length));
   b = copy (normalize(b, a.length));

   // Zadbaj, aby a było większe od b; zwróć GCD w banalnym przypadku
   int rc = compareTo(a, b);
   if (rx == 0) { assign (gcd, a); return; }
   if (rc < 0) {
      int [] t = b;
      b = a;
      a = t;
   }

   int [] quot = new int[a.length];
   int [] remainder = new int[a.length];
```

```
    while(!isZero(b)) {
       int [] t = copy (b);
       divide (a, b, quot, remainder);
       assign (b, remainder);
       assign (a, t);
    }

    // Wartość przechowywana w a jest obliczonym GCD liczb (a, b)
    assign (gcd, a);
}
```

MODGCD będzie dochodził do rozwiązania znacznie szybciej, gdyż nie marnuje czasu na odejmowanie nieraz naprawdę małych liczb od dużych liczb w pętli `while`. Ta różnica nie jest tylko szczegółem implementacyjnym; wyraża zasadniczą zmianę w sposobie potraktowania problemu w algorytmie.

Czasy obliczeń wykreślone na rysunku 2.7 (i wykazane liczbowo w tabeli 2.5) obrazują wyniki wygenerowania 142 losowych liczb *n*-cyfrowych i obliczenia największego wspólnego dzielnika każdej z 10 011 par tych liczb.



Rysunek 2.7. Porównanie gcd i modgcd

Mimo że realizacja MODGCD przewyższa odpowiednią realizację GCD prawie o 60%, sprawność MODGCD jest *kwadratowa*, czyli $O(n^2)$, natomiast GCD jest wykładniczy. Oznacza to, że najgorszy przypadek realizacji GCD (nie pokazany w naszym małym zbiorze wejściowym) jest o rzędy wielkości gorszy niż najgorszy przypadek MODGCD.

Obmyślono bardziej wyrafinowane algorytmy obliczania GCD, choć większość z nich jest niepraktyczna, wyjąwszy przypadki bardzo dużych liczb. Z analizy wynika również, że problem ten umożliwia zbudowanie wydajniejszych algorytmów.

Tabela 2.5. Czas (w milisekundach) wykonania 10 011 obliczeń GCD

n	modgcd	gcd	n^2/modgcd	n^2/gcd	$\text{modgcd}_{2n}/\text{modgcd}_n$	$\text{gcd}_{2n}/\text{gcd}_n$
2	234	62	0,017	0,065		
4	391	250	0,041	0,064	1,67	4,03
8	1046	1984	0,061	0,032	2,68	7,94
16	2953	6406	0,087	0,040	2,82	3,23
32	8812	18 609	0,116	0,055	2,98	2,90
64	29 891	83 921	0,137	0,049	3,39	4,51
128	106 516	321 891	0,154	0,051	3,56	3,84

Mieszanka działań

Jak opisano wcześniej, w ramce „Wpływ kodowania na sprawność", projektant musi uwzględnić wiele operacji naraz. Nie wszystkie operacje można optymalizować; w rzeczywistości zoptymalizowanie jednej może pogorszyć działanie innej. Jako przykład rozważmy strukturę danych z operacjami *op1* i *op2*. Załóżmy, że istnieją dwa sposoby zrealizowania tej struktury: *A* i *B*. Na potrzeby naszych rozważań przyjmiemy, że nie musimy nic wiedzieć o tej strukturze danych ani o żadnym z tych sposobów. Budujemy dwa scenariusze:

Małe zbiory danych
: Przyjmując rozmiar n = 1000 elementów, zmieszaj 2000 operacji *op1* z 3000 operacji *op2*.

Duże zbiory danych
: Przyjmując rozmiar n = 100 000 elementów, zmieszaj 200 000 operacji *op1* z 300 000 operacji *op2*.

Tablica 2.6 zawiera oczekiwane wyniki wykonania realizacji *A* i *B* na tych dwu zestawach danych. Z pierwszego wiersza tabeli widać, że średni koszt wykonania *op1* w realizacji *A* na danych o rozmiarze n = 1000 wyniósł szacunkowo 0,008 ms; pozostałe wartości w drugiej i trzeciej kolumnie należy interpretować podobnie. W ostatniej kolumnie podano sumaryczny oczekiwany czas wykonania; zatem dla wariantu *A* i danych rozmiaru n = 1000 oczekujemy czasu 2000·0,008+3000·0,001 = 16+3 = 19 milisekund. Choć realizacja *B* początkowo prześciga realizację *A* dla małych wartości n, sytuacja zmienia się diametralnie, gdy skala problemu zwiększy się o dwa rzędy wielkości. Zauważmy, że wariant *A* jest dobrze skalowalny, natomiast *B* zachowuje się okropnie.

Tabela 2.6. Porównanie operacji z różnych realizacji

Rozmiar wejścia	op1 (ms)	op2 (ms)	#op1	#op2	Suma (ms)
A na 1000	0,008	0,001	2000	3000	19
A na 100 000	0,0016	0,003	200 000	300 000	1220
B na 1000	0,001	0,001	2000	3000	5
B na 100 000	0,1653	0,5619	200 000	300 000	201 630

Operacje do pomiarów wzorcowych

Program w języku Scheme w przykładzie 2.7 oblicza 2^n; poniżej pokazano przykład obliczenia 2^{851}.

Przykład 2.7. Kosztowne obliczenia

```
;; Dwa do n-tej: liczba -> liczba
(define (TwoToTheN n)
   (let loop ([i n]
              [result 1])
      if (= i 0)
         result
         (loop (sub1 i) (* 2 result)))))

;; Wynik przykładowego obliczenia
(TwoToTheN 851)
150150336576094004599423153910185137226235191870990070733557987815252631252384634158948203971606627616971080383694109252383653813326044865235229218132798103200794538451818051546732566997782908246399595358358052523086606780893692342385292277744791953321492480
```

W języku Scheme obliczenia są względnie niezależne od platformy. Otóż obliczenie 2^{851} w Javie lub C na większości platform spowodowałoby nadmiar numeryczny[11]. Lecz szybkie obliczenie w Scheme daje wynik pokazany w przykładzie. Czy zatem ukrywanie właściwości platformy, abstrahowanie od niej jest zaletą, czy wadą? Rozważmy dwie następujące hipotezy:

Hipoteza H1
: Obliczenie 2^n zachowuje się jednolicie niezależnie od wartości n.

Hipoteza H2
: Wielkie liczby (jak pokazana w poprzednim rozwinięciu) można traktować tak samo jak każdą inną liczbę, na przykład 123 827 lub 997.

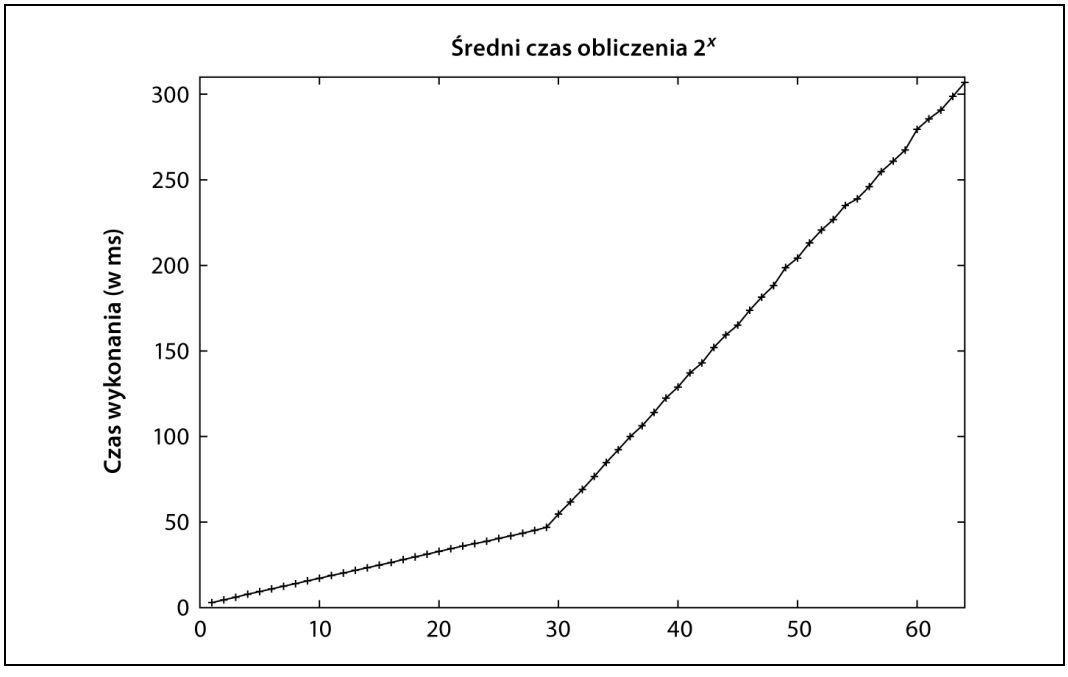
Aby odrzucić hipotezę H1, wykonujemy 50 prób, w których obliczamy 10 000 potęg 2^n. Pomijamy najlepszy i najgorszy rezultat, pozostawiając 48 prób. Średni czas wykonania tych 48 prób pokazano na rysunku 2.8.

Na początku wyraźnie widać zależność liniową występującą przy zwiększaniu liczby operacji mnóż-przez-2. Gdy jednak wartość x dojdzie do około 30, wystąpi inna zależność liniowa. Z jakichś powodów sprawność obliczania zmienia się, gdy zacznie się używać potęg 2 większych niż 30.

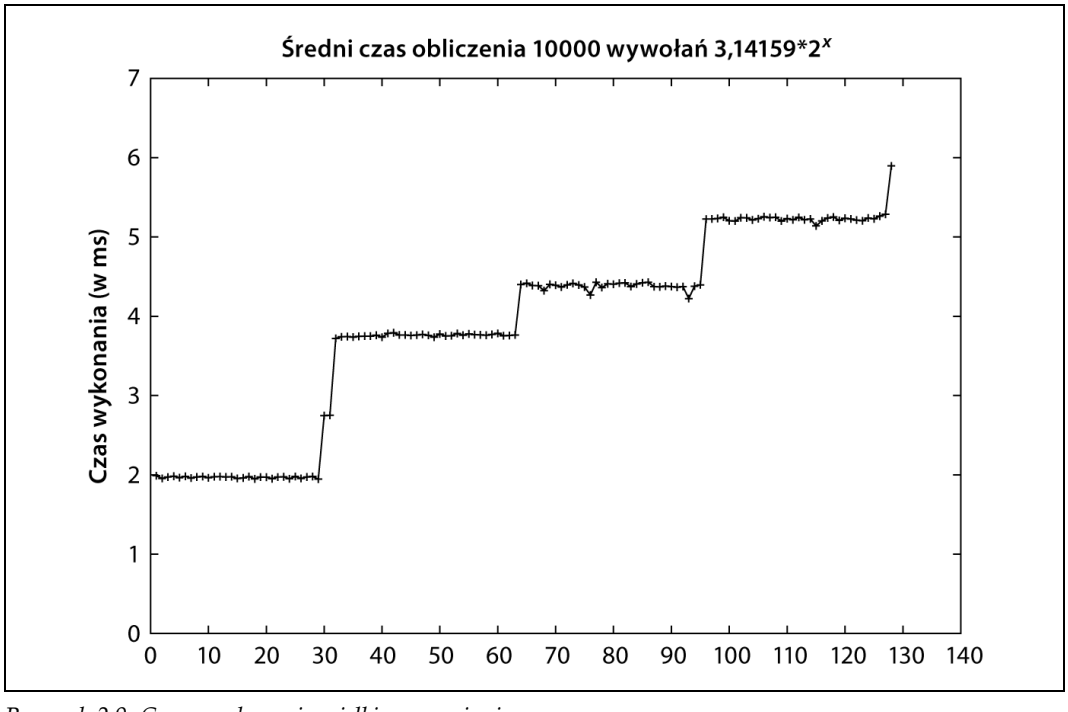
Żeby odrzucić hipotezę H2, wykonujemy eksperyment, w którym najpierw jest obliczana wartość 2^n, a potem jest wyznaczany czas obliczenia $3{,}14159 \cdot 2^n$. Wykonaliśmy 50 prób, a w każdej z nich po 10 000 obliczeń wartości $3{,}14159 \cdot 2^n$. Pominęliśmy wynik najlepszy i najgorszy, zostawiając wyniki 48 prób. Średni czas tych 48 prób jest przedstawiony na rysunku 2.9 (wyniki te są zasadniczo takie same nawet wówczas, gdy zamiast przez 3,14159 mnożymy przez 1,0000001).

Dlaczego punkty na rysunku 2.9 nie układają się w linii prostej? Dla jakiej wartości x linia ta się załamuje? Można odnieść wrażenie, że operacja mnożenia ($\cdot$) jest dociążana. Wykonuje coś innego, zależnie od tego, czy mnożone liczby są zmiennopozycyjne, czy całkowite mieszczące się w jednym słowie maszynowym, czy też całkowite, lecz tak duże, że muszą być pamiętane w kilku słowach maszyny, lub są kombinacją tychże.

[11] Scheme, odmiana Lispu, działa interpretacyjnie. Jeżeli zaprogramować to obliczenie w sposób interpretacyjny, na przykład na tablicach rezerwowanych statycznie lub dynamicznie, to nie ma przeszkód, aby wykonać je w tych językach — *przyp. tłum.*



Rysunek 2.8. Czasy obliczania 2^x



Rysunek 2.9. Czasy wykonania wielkiego mnożenia

Pierwszy skok na wykresie występuje dla $x = \{30, 31\}$, co nie jest łatwe do wyjaśnienia. Pozostałe płaskowyże można wyjaśnić bardziej konwencjonalnie, ponieważ występują przy wartościach (32, 64, 96, 128), co ma związek z długością słowa komputera, na którym wykonywano próby (mianowicie: jedno, dwa, trzy lub cztery słowa 32-bitowe). W miarę jak do zapamiętania liczby trzeba coraz więcej słów, wzrasta też czas potrzebny do wykonania mnożenia.

Operacja (kwalifikująca się) do pomiarów wzorcowych (ang. *benchmark operation*) ma zasadnicze znaczenie w algorytmie, jako że zliczanie czasów jej wykonania stanowi dobrą prognozę czasu wykonania programu. Operacją do pomiarów wzorcowych w `TwoToTheN` jest ·, czyli operacja mnożenia.

Uwaga końcowa

W tej książce uprościliśmy omówienie notacji „duże O". Na przykład, omawiając *liniowość* zachowania algorytmu DODAWANIE względem rozmiaru wejścia n, powiedzieliśmy, że istnieje pewna stała $c > 0$, taka że $t(n) \leq c \cdot n$ dla każdego $n > n_0$; przypomnijmy, że $t(n)$ oznacza faktyczny czas wykonania DODAWANIA. Wnioskując w ten sposób, oświadczamy, że sprawność DODAWANIA wynosi $O(n)$. Uważny Czytelnik dostrzeże, że równie dobrze moglibyśmy użyć funkcji $f(n) = c \cdot 2^n$, która rośnie znacznie szybciej niż $c \cdot n$. Rzeczywiście, choć z technicznego punktu widzenia można by orzec, że DODAWANIE ma złożoność $O(2^n)$, stwierdzenie takie dostarczyłoby bardzo mało informacji (to tak, jakby powiedzieć, że na wykonanie 5-minutowego zadania trzeba Ci nie więcej niż tydzień). Aby zdać sobie z tego sprawę, weźmy pod uwagę notację $\Omega(g(n))$, która symbolizuje, że $g(n) \leq t(n)$ jest dolnym ograniczeniem rzeczywistego czasu wykonania. Często można obliczyć zarówno górne (O), jak i dolne (Ω) ograniczenie czasu wykonania algorytmu, a wówczas używa się stosownej notacji $\Theta(f(n))$, z którą wiąże się założenie, że $f(n)$ jest asymptotycznie zarówno górnym (co odpowiada przypadkowi $O(f(n))$, jak i dolnym ograniczeniem $t(n)$.

Wybieramy mniej formalne (i powszechnie akceptowane) użycie $O(f(n))$, aby uprościć prezentacje i analizy. Gwarantujemy, że jeśli podczas omawiania zachowania algorytmicznego sklasyfikujemy złożoność jakichś algorytmów jako $O(f(n))$, to w odniesieniu do nich nie ma lepszej od $f(n)$ funkcji $f'(n)$.

Literatura

Jon Louis Bentley, M. Douglas McIlroy: *Engineering a Sort Function*. „Software — Practice and Experience" 1993, 23, 11, s. 1249 – 1265, *http://citeseer.ist.psu.edu/bentley93engineering.html*.

D. Zuras: *More on Squaring and Multiplying Large Integers*. „IEEE Transactions on Computers" 1994, 43, 8, s. 899 – 908, *http://doi.ieeecomputersociety.org/10.1109/12.295852*.