



Perełki programowania

Wydanie II

PEREŁKA WŚRÓD KSIĄŻEK O PROGRAMOWANIU!

- Jak projektować algorytm?
- Jak oszacować i zmierzyć wydajność algorytmu?
- Jak skompresować kod programu oraz dane?

Jon Bentley

Tytuł oryginału: Programming Pearls (2nd Edition)

Tłumaczenie: Ireneusz Jakóbiak

ISBN: 978-83-246-3481-1

Authorised translation from the English language edition, entitled: Programming Pearls, 2nd edition ISBN 0201657880, by Jon Bentley, published by Pearson Education, Inc, publishing as Addison Wesley, Copyright © 2000 by Lucent Technologies, known as Alcatel-Lucent USA, Inc. Originally published as part of ACM Press Books, a collaboration between the Association Computing Machinery and Addison-Wesley

Polish language edition published by Helion S.A.
Copyright © 2011

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 Pearson Education Inc.

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

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

Autor oraz 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/perop2.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/perop2>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Wstęp	7
I Preliminaria	11
1. Rozłupywanie ostrygi	13
1.1. Przyjacielska rozmowa	13
1.2. Dokładne określenie problemu.....	14
1.3. Projekt programu	14
1.4. Szkic implementacji	16
1.5. Reguły	17
1.6. Problemy	18
1.7. Lektury uzupełniające	19
2. Aha! Algorytmy	21
2.1. Trzy problemy	21
2.2. Wszędobylskie szukanie binarne	22
2.3. Potęga typów podstawowych	23
2.4. Składanie razem: sortowanie	25
2.5. Reguły	26
2.6. Problemy	27
2.7. Lektury uzupełniające	28
2.8. Implementacja programu anagramowego (kolumna boczna)	29
3. Dane strukturyzują programy	31
3.1. Program kwestionariuszowy	31
3.2. Programowanie listów seryjnych	33
3.3. Tablica przykładów	35
3.4. Strukturyzacja danych	37
3.5. Wydajne narzędzie dla wyspecjalizowanych danych	37
3.6. Reguły	39
3.7. Problemy	40
3.8. Lektury uzupełniające	42

4. Pisanie poprawnych programów	43
4.1. Wyzwanie szukania binarnego	43
4.2. Pisanie programu	44
4.3. Zrozumieć program	46
4.4. Reguły	49
4.5. Zadania weryfikacji oprogramowania	50
4.6. Problemy	51
4.7. Lektury uzupełniające	53
5. Drobną kwestia zaprogramowania	55
5.1. Od pseudokodu do C	55
5.2. Zaprzęg testowy	57
5.3. Sztuka asercji	59
5.4. Testowanie zautomatyzowane	60
5.5. Pomiary prędkości	61
5.6. Gotowy program	63
5.7. Reguły	63
5.8. Problemy	64
5.9. Lektury uzupełniające	65
5.10. Debugowanie (kolumna boczna)	66
II Wydajność	69
6. Spojrzenie na wydajność	71
6.1. Studium przypadku	71
6.2. Poziomy projektowania	73
6.3. Reguły	75
6.4. Problemy	75
6.5. Lektury uzupełniające	76
7. Obliczenia pobieżne	77
7.1. Podstawowe umiejętności	78
7.2. Szacowanie wydajności	80
7.3. Czynniki bezpieczeństwa	82
7.4. Prawo Little'a	83
7.5. Reguły	84
7.6. Problemy	84
7.7. Lektury uzupełniające	85
7.8. Obliczenia pobieżne w życiu codziennym	86
8. Techniki projektowania algorytmów	87
8.1. Problem i prosty algorytm	87
8.2. Dwa algorytmy kwadratowe	88
8.3. Algorytm dziel i zwyciężaj	89
8.4. Algorytm skanujący	91
8.5. Jak to ma znaczenie?	92
8.6. Reguły	93
8.7. Problemy	94
8.8. Lektury uzupełniające	96

9. Optymalizacja kodu	97
9.1. Typowa historia	97
9.2. Zestaw pierwszej pomocy	98
9.3. Poważna operacja — szukanie binarne	102
9.4. Reguły	105
9.5. Problemy	107
9.6. Lektury uzupełniające	108
10. Oszczędzanie miejsca	109
10.1. Kluczem jest prostota	109
10.2. Problem poglądowy	110
10.3. Techniki zmniejszające wielkość danych	113
10.4. Techniki zmniejszające wielkość kodu	116
10.5. Reguły	118
10.6. Problemy	119
10.7. Lektury uzupełniające	120
10.8. Duża oszczędność (kolumna boczna)	120
III Produkt	123
11. Sortowanie	125
11.1. Sortowanie przez wstawianie	125
11.2. Proste sortowanie szybkie	127
11.3. Lepsze szybkie sortowania	130
11.4. Reguły	132
11.5. Problemy	133
11.6. Lektury uzupełniające	134
12. Problem próby	137
12.1. Problem	137
12.2. Jedno rozwiązanie	138
12.3. Przestrzeń projektowania	139
12.4. Reguły	142
12.5. Problemy	143
12.6. Lektury uzupełniające	144
13. Szukanie	145
13.1. Interfejs	145
13.2. Struktury liniowe	147
13.3. Binarne drzewa poszukiwań	150
13.4. Struktury dla liczb całkowitych	153
13.5. Reguły	155
13.6. Problemy	156
13.7. Lektury uzupełniające	156
13.8. Rzeczywisty problem szukania (kolumna boczna)	157
14. Kopce	161
14.1. Struktura danych	161
14.2. Dwie krytyczne funkcje	163

14.3. Kolejki priorytetowe	166
14.4. Algorytm sortujący	169
14.5. Reguły	171
14.6. Problemy	171
14.7. Lektury uzupełniające	173
15. Sznury pereł	175
15.1. Słowa	175
15.2. Frazy	178
15.3. Generowanie tekstu	181
15.4. Reguły	185
15.5. Problemy	186
15.6. Lektury uzupełniające	187
Zakończenie do wydania pierwszego	189
Zakończenie do wydania drugiego	191
A. Katalog algorytmów	193
B. Quiz estymacyjny	199
C. Modele kosztowe czasu i pamięci	201
D. Reguły optymalizacji kodu	207
E. Klasy języka C++ służące do szukania	213
Wskazówki do wybranych problemów	217
Rozwiązania wybranych problemów	223
Skorowidz	251

Optymalizacja kodu

Niektórzy programiści zwracają zbyt wielką uwagę na wydajność; wprowadzając za szybko niewielkie „optymalizacje”, tworzą niezmiernie sprytnie programy, które są bardzo trudne w konserwacji. Inni zwracają zbyt mało uwagi i uzyskują pięknie ustrukturalizowane programy, które są wysoce niewydajne i w związku z tym bezużyteczne. Dobrzy programiści mają wydajność na uwadze: jest ona jednak tylko jednym z aspektów oprogramowania, chociaż czasami bardzo ważnym.

W poprzednich rozdziałach omawiano wysokopoziomowe podejście do wydajności: definowanie problemu, strukturę systemu, projektowanie algorytmu i wybór struktury danych. Bieżący rozdział jest poświęcony podejściu niskopoziomowemu. „Optymalizacja kodu” pozwala na zlokalizowanie kosztownych fragmentów w istniejącym programie i wprowadzenie w nim niewielkich zmian zwiększających prędkość działania. Nie zawsze jest to właściwe podejście i rzadko bywa efektywne, ale czasami może przynieść ogromną poprawę wydajności systemu.

9.1. Typowa historia

Pewnego wczesnego popołudnia gawędziłem z Chrisem Van Wykiem na temat optymalizacji kodu. Po naszej rozmowie Chris poszedł udoskonalić program w C. Kilka godzin później skrócił o połowę czas działania programu graficznego składającego się z trzech tysięcy wierszy.

Chociaż czas pracy przy typowych obrazach był znacznie krótszy, program potrzebował dziesięciu minut na przetworzenie bardziej skomplikowanych obrazów. Pierwszym krokiem Van Wyka było **sprofilowanie** programu, aby dowiedzieć się, ile czasu jest poświęcane na każdą funkcję (profil podobnego, chociaż mniejszego programu pokazano na następnej stronie). Uruchomienie programu dla dziesięciu typowych obrazów testowych pokazało, że poświęcał on prawie siedemdziesiąt procent swojego czasu na wykonywanie funkcji alokacji pamięci `malloc`.

Następnym krokiem Van Wyka było przestudiowanie alokatora pamięci. Ponieważ jego program wywoływał instrukcję `malloc` za pośrednictwem jednej funkcji, która udostępniała kontrolę błędów, mógł modyfikować tę funkcję bez sprawdzania kodu źródłowego instrukcji `malloc`. Van Wyk dodał kilka wierszy kodu zliczającego, co pokazało, że najpopularniejszy rodzaj rekordu był alokowany trzydzieści razy częściej niż następny pod względem popularności rodzaj. Wiedząc, że program poświęca większość swojego czasu na szukanie w pamięci jednego typu rekordu, jak mógłbyś zmodyfikować program, aby działał szybciej?

Van Wyk rozwiązał swój problem, stosując zasadę *zapisywania w pamięci podręcznej*: dane, które są potrzebne najczęściej, powinny być najmniej kosztowne do odczytania. Zmodyfikował on program, umieszczając w pamięci podręcznej wolne rekordy najpopularniejszych typów na liście powiązanej. Mógł dzięki temu obsłużyć typowe zapytanie poprzez szybkie sięgnięcie do tej listy, zamiast wywoływać alokator pamięci ogólnego przeznaczenia. Pozwoliło to skrócić łączny czas działania programu o 45 procent w porównaniu z tym, jak działał wcześniej (tak więc praca alokatora pamięci zabierała obecnie 30 procent łącznego czasu). Dodatkową korzyścią było też to, że obniżona fragmentacja zmodyfikowanego alokatora spowodowała wydajniejsze korzystanie z pamięci głównej w porównaniu z wcześniejszym alokatorem. W rozwiązaniu 2. pokazano alternatywną implementację tej starożytnej techniki. Skorzystamy jeszcze z podobnego podejścia kilka razy w rozdziale 13.

Historia ta ilustruje sztukę optymalizacji kodu w najlepszym wydaniu. Poświęcając kilka godzin na pomiary i dodanie około dwudziestu wierszy do liczącego 3000 linii programu, Van Wyk podwoił jego prędkość bez wprowadzania zmian w częściach widocznych dla użytkownika i bez zwiększania stopnia trudności jego obsługi. Aby uzyskać wzrost prędkości, skorzystał z narzędzi ogólnego przeznaczenia: profilowanie zidentyfikowało „punkt zapalny” w jego programie, a użycie pamięci podręcznej skróciło czas działania.

Poniższy profil typowego niewielkiego programu w C jest podobny — zarówno w formie, jak i zawartości — do profilu Van Wyka:

Czas działania funkcji	%	Czas działania funkcji i jej potomnych	%	Liczba wywołań	Funkcja
1413,406	52,8	1413,406	52,8	200002	malloc
474,441	17,7	2109,506	78,8	200180	insert
285,298	10,7	1635,065	61,1	250614	rinsert
174,205	6,5	2675,624	100	1	main
157,135	5,9	157,135	5,9	1	report
143,285	5,4	143,285	5,4	200180	bigrand
27,854	1,0	91,493	3,4	1	initbins

W tym przebiegu większość czasu była poświęcana funkcji malloc. Problem 2. zachęca do skrócenia czasu działania tego programu przez zapisywanie węzłów w pamięci podręcznej.

9.2. Zestaw pierwszej pomocy

Przeniesimy teraz naszą uwagę z dużego programu na kilka małych funkcji. Każda z nich dotyczy problemu, z którym zetknąłem się w różnych kontekstach. Problemy zabierały większość czasu działania aplikacji, a w rozwiązaniach użyto reguł ogólnego zastosowania.

Problem nr jeden — reszta z dzielenia całkowitego. W podrozdziale 2.3 zarysowano trzy algorytmy służące do rotowania wektorów. W rozwiązaniu 2.3 zaimplementowano algorytm „żonglujący” z następującym poleceniem w pętli wewnętrznej:

```
k = (j + rotdist) % n;
```

Model kosztów w dodatku C pokazuje, że w języku C operator reszty z dzielenia % może być bardzo kosztowny; podczas gdy większość operacji arytmetycznych potrzebuje około dziesięciu nanosekund, operator % zabiera prawie 100 nanosekund. Moglibyśmy skrócić czas działania programu, implementując operację dzielenia za pomocą następującego kodu:

```
k = j + rotldist;
if (k >= n)
    k -= n;
```

Rozwiązanie to zastępuje kosztowny operator % porównaniem i (rzadko) odejmowaniem. Czy daje ono jednak jakąś różnicę w działaniu całej funkcji?

W pierwszym eksperymencie uruchomiłem program z odległością rotacji `rotldist` ustawioną na 1 i czas działania spadł z 119n nanosekund do 57n nanosekund. Program był niemal dwukrotnie szybszy, a zaobserwowane 62 nanosekundy przyspieszenia były zbliżone do przewidywań wynikających z modelu kosztów.

W następnym eksperymencie zmieniłem wartość `rotldist` na 10 i zaskoczony stwierdziłem, że czas pracy programu w obu metodach był identyczny i wyniósł 206n nanosekund. Eksperymenty takie jak sporządzenie wykresu dla rozwiązania 2.4 wkrótce pozwoliły mi określić przyczynę: dla `rotldist` równego 1 algorytm odwoływał się do pamięci sekwencyjnie i operator reszty z dzielenia zdominował czas działania programu. Kiedy jednak `rotldist` wynosiło 10, kod odwoływał się do co dziesiątego słowa pamięci i dominowało przenoszenie zawartości RAM-u do pamięci podręcznej.

W starych czasach programiści nauczyli się, że próby przyspieszenia obliczeń w programach, które większość czasu poświęcały na operacje wejścia i wyjścia, są bezcelowe. W nowoczesnych architekturach równie bezcelowe jest podejmowanie prób skrócenia czasu obliczeń, kiedy tak wiele cykli jest poświęcanych uzyskiwaniu dostępu do pamięci.

Problem nr dwa — funkcje, makra i kod inline. W całym rozdziale 8. musieliśmy obliczać maksimum dwóch wartości. Na przykład w podrozdziale 8.4 korzystaliśmy z takiego kodu jak:

```
maxendinghere = max(maxendinghere, 0);
maxsofar = max(maxsofar, maxendinghere);
```

Funkcja `max` zwraca większą wartość spośród jej dwu argumentów:

```
float max(float a, float b)
{    return a > b ? a : b; }
```

Czas działania tego programu wynosi około 89n nanosekund.

Starzy programiści pracujący w C mogą zareagować odruchowo i zastąpić tę funkcję następującym makrem:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Jest ono zdecydowanie brzydsze i bardziej podatne na błędy. W przypadku wielu optymalizujących kompilatorów zastosowanie tego makra nie wywoła zupełnie żadnej różnicy (takie kompilatory zapisują małe funkcje *inline*). W moim systemie jednak taka zmiana spowodowała skrócenie czasu działania algorytmu 4 z 89n nanosekund do 47n nanosekund, co daje prawie dwukrotny wzrost prędkości.

Mój zachwyt nad takim przyspieszeniem wyparował, kiedy zmierzyłem efekt wprowadzenia makra do algorytmu 3 z podrozdziału 8.3. Dla n równego 10 000 czas działania wzrósł z dziesięciu milisekund do stu sekund, czyli program spowolnił 10 000 razy. Makro zdawało się wydłużać czas działania algorytmu 3 z poprzedniego $O(n \log n)$ do wartości bliższej $O(n^2)$. Wkrótce odkryłem, że semantyka wywoływania makr po nazwie sprawiła, że algorytm 3 wywoływał się rekursywnie więcej niż dwukrotnie, co asymptotycznie wydłużyło czas działania. W problemie 4. pokazano bardziej drastyczny przykład tego typu spowolnienia.

Podczas gdy programujący w C muszą się martwić kompromisami między wydajnością a poprawnością, programujący w C++ mogą cieszyć się tym, co najlepsze w jednym i drugim. W C++ można wskazać, że funkcja ma być skompilowana inline, co łączy czystą semantykę funkcji z małym narzutem korzystania z makr.

Z czystej ciekawości pominąłem zarówno makra, jak również funkcje, i zaprogramowałem obliczenia za pomocą instrukcji `if`:

```
if (maxendinghere < 0)
    maxendinghere = 0;
if (maxsofar < maxendinghere)
    maxsofar = maxendinghere
```

Czas działania pozostał w zasadzie niezmienny.

Problem nr trzy — szukania sekwencyjne. Teraz skupimy się na sekwencyjnym przeszukiwaniu (potencjalnie nieposortowanej) tablicy:

```
int ssearch1(t)
for i = [0, n)
    if x[i] == t
        return i
return -1
```

Ten zwięzły kod potrzebuje przeciętnie $4,06n$ nanosekund, aby odszukać element znajdujący się w tablicy `x`. Ponieważ przy typowym udanym szukaniu sprawdza on tylko połowę elementów tablicy, każdemu elementowi poświęca około 8,1 nanosekundy.

Pętla jest już smukła, ale można jeszcze od niej odkroić niewielki kawałek sadelka. Pętla wewnętrzna zawiera dwa testy: pierwszy z nich sprawdza, czy `i` znajduje się na końcu tablicy, a drugi, czy `x[i]` jest potrzebnym elementem. Pierwszy test można zastąpić drugim, umieszczając na końcu tablicy **strażnika**:

```
int ssearch2(t)
hold = x[n]
x[n] = t
for (i = 0; ; i++)
    if x[i] == t
        break
x[n] = hold
if i == n
    return -1
else
    return i
```

Zabieg ten skraca czas do $3,87n$ nanosekund, co oznacza przyspieszenie o około pięć procent. W kodzie założono, że tablica została alokowana w taki sposób, iż $x[n]$ można tymczasowo nadpisać. Rozsądne jest zapisanie elementu $x[n]$ i odzyskanie go po każdym szukaniu; zabieg taki jest jednak zbyteczny w większości aplikacji i zostanie usunięty w następnej wersji.

Najbardziej wewnętrzna pętla zawiera już tylko inkrementację, dostęp do tablicy i sprawdzenie. Czy można ją jeszcze bardziej zredukować? W ostatecznej wersji naszego programu szukającego pętla zostanie rozwinięta ośmiokrotnie w celu pozbycia się inkrementacji; dalsze rozwijanie nie przyczyni się do wzrostu prędkości.

```
int ssearch3(t)
  x[n] = t
  for (i = 0; ; i += 8)
    if (x[i] == t) { break }
    if (x[i+1] == t) { i += 1; break }
    if (x[i+2] == t) { i += 2; break }
    if (x[i+3] == t) { i += 3; break }
    if (x[i+4] == t) { i += 4; break }
    if (x[i+5] == t) { i += 5; break }
    if (x[i+6] == t) { i += 6; break }
    if (x[i+7] == t) { i += 7; break }
  if i == n
    return -1
  else
    return i
```

Zmiana ta skraca czas do $1,70n$ nanosekund, co oznacza przyspieszenie o około 56 procent. Na starszych komputerach redukcja narzutu mogłaby dać wzrost prędkości o 10 lub 20 procent, choć i w nowych maszynach rozwijanie pętli może być pomocne w unikaniu zatorów potoków, zmniejszeniu liczby rozgałęzień i zwiększeniu paralelizmu na poziomie instrukcji.

Problem nr cztery — obliczanie odległości sferycznych. Ostatni problem jest typowy dla zastosowań, w których wykorzystywane są dane geograficzne lub geometryczne. Pierwszą częścią danych wejściowych jest zbiór S pięciu tysięcy punktów na powierzchni sfery. Każdy punkt jest opisany przez szerokość i długość geograficzną. Po tym, jak punkty te zostaną zapisane w strukturze danych naszego wyboru, program wczyta drugą część danych wejściowych: sekwencję dwudziestu tysięcy punktów także opisanych szerokością i długością geograficzną. Dla każdego punktu z tej sekwencji program musi określić, który punkt ze zbioru S jest względem niego najbliższy, przy czym odległość jest wyrażona wielkością kąta tworzonego przez promienie biegnące ze środka sfery do tych dwu punktów.

Margaret Wright napotkała podobny problem, pracując na Uniwersytecie Stanfordzkim we wczesnych latach osiemdziesiątych, gdy na podstawie map sumowała dane dotyczące globalnego rozkładu pewnych cech genetycznych. W rozwiązaniu bezpośrednim zbiór S był reprezentowany przez tablicę z szerokościami i długościami geograficznymi. Najbliższy sąsiad każdego punktu z sekwencji był odnajdywany poprzez obliczenie odległości do poszczególnych punktów w S za pomocą skomplikowanego wzoru trygonometrycznego zawierającego dziesięć funkcji sinus i kosinus. Chociaż program był prosty do zakodowania i tworzył dobre mapy dla małych zbiorów danych, to każda duża mapa wymagała już wielu godzin pracy komputera dużej mocy, co było zdecydowanie poza budżetem całego projektu.

Ponieważ pracowałem wcześniej nad problemami geometrycznymi, Wright poprosiła mnie, żebym spróbował swoich sił także i tu. Po spędzeniu nad problemem większości weekendu opracowałem kilka wyszukanych algorytmów i struktur danych, które mogłyby go rozwiązać. Na szczęście (z perspektywy czasu) każdy z nich wymagałby wielu setek wierszy kodu, przez co nawet nie próbowałem ich zaimplementować. Kiedy opisałem moje struktury danych Andrew Appelowi, wpadł on na genialny pomysł: zamiast zabierać się za problem na poziomie struktur danych, czemu nie użyć prostej struktury w postaci tablicy przechowującej punkty, ale za to zoptymalizować kod w celu obniżenia kosztu obliczania odległości między punktami? W jaki sposób skorzystałbyś z tego pomysłu?

Koszt można zredukować w znacznym stopniu, zmieniając reprezentację punktów. Zamiast korzystać z szerokości i długości, przedstawimy lokalizację punktów na powierzchni sfery za pomocą współrzędnych x , y i z . Tak więc struktura danych będzie tablicą przechowującą szerokość i długość geograficzną każdego punktu (które mogą być potrzebne przy innych operacjach) oraz ich trzy współrzędne kartezjańskie. Podczas przetwarzania każdego punktu sekwencji kilka funkcji trygonometrycznych przełoży szerokość i długość na współrzędne x , y i z , a następnie zostanie policzona ich odległość do poszczególnych punktów ze zbioru S . Odległości te będą liczone jako sumy kwadratów różnic w trzech wymiarach, co jest zwykle o wiele mniej kosztowne niż obliczanie wartości funkcji trygonometrycznej, nie mówiąc już o dziesięciu (w modelu kosztów prędkości działania w dodatku C pokazano szczegółowe informacje dla jednego systemu). Metoda ta daje poprawną odpowiedź, ponieważ kąt zawarty między dwoma punktami zwiększa się monotonicznie wraz z kwadratem euklidesowej odległości między nimi.

Chociaż podejście to wymaga dodatkowej pamięci, przynosi ono wymierne korzyści: kiedy Wright wprowadziła zmiany w swoim programie, czas jego działania w przypadku złożonych map spadł z kilku godzin do połowy minuty. W tym przypadku optymalizacja kodu rozwiązała problem za pomocą kilkudziesięciu linii programu, podczas gdy zmiany w algorytmie i strukturze danych wymagałyby setek wierszy kodu.

9.3. Poważna operacja — szukanie binarne

Teraz skierujemy naszą uwagę w stronę jednego z najbardziej ekstremalnych przypadków optymalizacji kodu, jakie znam. Szczegóły pochodzą z problemu 4.8: chcemy przeszukać binarnie tablicę ze stoma tysiącami liczb całkowitych. Kiedy będziemy omawiać proces, pamiętaj, że ulepszanie szukania binarnego zazwyczaj nie jest konieczne — algorytm jest na tyle wydajny, że optymalizacja kodu często okazuje się zbyteczna. W rozdziale 4. zignorowaliśmy zatem mikroskopijną wydajność i skupiliśmy się na osiągnięciu prostego, poprawnego i łatwego w obsłudze programu. Czasami jednak zoptymalizowane szukanie może sprawiać w systemie ogromną różnicę.

Opracujemy szybkie szukanie binarne w sekwencji czterech programów. Są one subtelne, ale istnieje dobry powód, aby je śledzić z uwagą: końcowy program będzie zwykle dwa do trzech razy szybszy od szukania binarnego z podrozdziału 4.2. Zanim zaczniesz czytać dalej, pomyśl, czy potrafisz wskazać oczywiste marnotrawstwo w kodzie?

```
l = 0; u = n-1
loop
  /* niezmiennik: jeśli t jest obecne, znajduje się w x[l..u] */
  if l > u
```

```

    p = -1; break
m = (l + u) / 2
case
    x[m] < t:  l = m+1
    x[m] == t: p = m; break
    x[m] > t:  u = m-1

```

Pracę nad szybkim szukaniem binarnym zaczniemy od zmodyfikowanego problemu polegającego na zlokalizowaniu *pierwszego* wystąpienia liczby całkowitej t w tablicy $x[0..n-1]$. Powyższy kod mógłby zwracać dowolne spośród wielu wystąpień t (właśnie takiego szukania będziemy potrzebować w podrozdziale 15.3). Główna pętla programu jest podobna do pętli z kodu powyżej; indeksy l i u będziemy przechowywać w tablicy ograniczającej położenie elementu t , ale skorzystamy z niezmienniej relacji, zgodnie z którą $x[l] < t \leq x[u]$ oraz $l < u$. Założymy, że $n \geq 0$, $x[-1] < t$ i że $x[n] \geq t$ (choć program nigdy nie sięgnie po te dwa fikcyjne elementy). Kod szukania binarnego wygląda teraz następująco:

```

l = -1; u = n
while l+1 != u
    /* niezmiennik: x[l] < t && x[u] >= t && l < u */
    m = (l + u) / 2
    if x[m] < t
        l = m
    else
        u = m
/* asercja l+1 = u && x[l] < t && x[u] >= t */
p = u
if p >= n || x[p] != t
    p = -1

```

Pierwszy wiersz inicjalizuje niezmiennik. Podczas powtarzania pętli niezmiennik pozostaje zachowany dzięki instrukcji `if`; łatwo sprawdzić, że oba rozgałęzienia zachowują niezmiennik. Po zakończeniu pętli wiemy, że jeśli element t w ogóle znajduje się w tablicy, to jego pierwsze wystąpienie jest na pozycji u . Fakt ten został wyrażony bardziej formalnie w komentarzu z asercją. Ostatnie dwie instrukcje ustawiają p na indeks pierwszego wystąpienia elementu t w tablicy x , jeśli element ten istnieje, natomiast na -1 , jeśli go nie ma.

Chociaż to szukanie binarne rozwiązuje nieco trudniejszy problem niż poprzedni program, jest ono potencjalnie wydajniejsze: dokonuje tylko jednego porównania elementu t z tablicą x w każdej iteracji pętli. Poprzedni program musiał czasami dokonywać dwóch takich porównań.

Następna wersja programu jako pierwsza skorzysta z tego, że wiemy, iż n wynosi 1000. Wykorzystano w niej inną reprezentację zakresu: zamiast przechowywać $l..u$ pod postacią dolnych i górnych wartości, zapamiętamy je jako dolną granicę l oraz przyrost i , taki że $l+i = u$. Kod zagwarantuje, że i zawsze będzie potęgą dwójki; właściwość ta będzie łatwa do zachowania, kiedy już ją ustanowimy, ale będzie trudna do uzyskania na wstępie (ponieważ tablica ma rozmiar $n = 1000$). Program zostanie zatem poprzedzony przypisaniem i instrukcją `if` zapewniającą, że przeszukiwany zakres ma początkowo rozmiar 512, czyli jest równy największej potędze dwójki mniejszej od 1000. Tak więc l oraz $l+i$ reprezentują teraz zakres $-1..511$ albo $488..1000$. Przepisanie poprzedniego programu szukania binarnego z uwzględnieniem nowej reprezentacji zakresu daje kod:

```

i = 512
l = -1
if x[511] < t
    l = 1000 - 512
while i != 1
    /* niezmiennik: x[l] < t && x[l+i] >= t && i = 2^j */
    nexti = l / 2
    if x[l+nexti] < t
        l = l + nexti
        i = nexti
    else
        i = nexti
/* asercja i == 1 && x[l] < t && x[l+i] >= t */
p = l + 1
if p > 1000 || x[p] != t
    p = -1

```

Przeprowadzenie dowodu poprawności tego programu ma dokładnie taki sam przebieg jak w przypadku wcześniejszego programu. Kod ten zwykle działa wolniej od swojego poprzednika, ale otwiera furtkę do kolejnych przyspieszeń.

Następny program jest uproszczeniem powyższego kodu, w którym uwzględniono kilka optymalizacji, jakich mógłby dokonać bystry kompilator. Pierwsza instrukcja `if` została uproszczona, zmienna `nexti` usunięta, a przypisania do `nexti` usunięte z wewnętrznej instrukcji `if`.

```

i = 512
l = -1
if x[511] < t
    l = 1000 - 512
while i != 1
    /* niezmiennik: x[l] < t && x[l+i] >= t && i = 2^j */
    i = i / 2
    if x[l+i] < t
        l = l + i
/* asercja i == 1 && x[l] < t && x[l+i] >= t */
p = l + i
if p > 1000 || x[p] != t
    p = -1

```

Chociaż uzasadnienie poprawności kodu nadal ma taką samą strukturę, możemy teraz zrozumieć jego działanie bardziej intuicyjnie. Kiedy pierwszy test zawiedzie i `l` pozostanie zerowe, program kolejno obliczy bity zmiennej `p`, począwszy od bitu najbardziej znaczącego.

Ostateczna wersja kodu nie jest przeznaczona dla słabych duchem. Usunięto w niej narzut związany ze sterowaniem pętlą oraz dzieleniem `i`, rozwijając pętlę. Ponieważ `i` przyjmuje w tym programie tylko dziesięć różnych wartości, możemy je wszystkie umieścić w kodzie, unikając tym samym konieczności ich ciągłego przeliczania podczas działania programu.

```

l = -1
if (x[511] < t) l = 1000 - 512
/* asercja x[l] < t && x[l+512] >= t */
if (x[l+256] < t) l += 256
/* asercja x[l] < t && x[l+256] >= t */
if (x[l+128] < t) l += 128

```

```

if (x[l+64 ] < t) l += 64
if (x[l+32 ] < t) l += 32
if (x[l+16 ] < t) l += 16
if (x[l+8  ] < t) l += 8
if (x[l+4  ] < t) l += 4
if (x[l+2  ] < t) l += 2
    /* asercja x[l] < t && x[l+2 ] >= t */
if (x[l+1  ] < t) l +=1
    /* asercja x[l] < t && x[l+1 ] >= t */
p = l+i
if p > 1000 || x[p] != t
    p = -1

```

Możemy uczynić ten kod bardziej zrozumiałym, wstawiając pełne komentarze z asercjami, takie jak otaczające wiersz z testem dla `x[l+256]`. Kiedy już przeprowadzimy analizę dwóch możliwych zachowań pierwszej instrukcji `if`, wszystkie pozostałe instrukcje `if` staną się zrozumiałe.

Porównałem działanie czystego szukania binarnego z podrozdziału 4.2 ze zoptymalizowanym szukaniem na różnych systemach. W pierwszym wydaniu tej książki przytoczyłem czasy działania programów na czterech różnych komputerach, w pięciu językach programowania i na kilku poziomach optymalizacji. Przyrosty prędkości wynosiły od 38 procent aż do przyspieszenia pięciokrotnego (skrócenie czasu o 80 procent). Kiedy dokonałem pomiarów na moim obecnym komputerze, zaszokowany, ale też i zachwycony stwierdziłem skrócenie czasu dla $n = 1000$ z 350 do 125 nanosekund na jedno szukanie (skrócenie czasu o 64 procent).

Przyspieszenie wyglądało zbyt pięknie, aby było prawdziwe, i tak też było w istocie. Bliższe badania wykazały, że moje rusztowanie pomiaru prędkości szukało każdego elementu tablicy po kolei: najpierw `x[0]`, potem `x[1]` itd. W ten sposób szukanie binarne uzyskało szczególnie korzystne wzorce dostępu do pamięci i doskonale przewidywalne rozgałęzienia. Zmieniłem w związku z tym rusztowanie w taki sposób, aby szukało elementów w kolejności losowej. Czyste szukanie binarne trwało tym razem 418 nanosekund, podczas gdy wersja z rozwiniętą pętlą potrzebowała 266 nanosekund, czyli wzrost prędkości wyniósł 36 procent.

Powyższy opis jest wyidealizowaną relacją z optymalizacji kodu w najbardziej ekstremalnej postaci. Zastąpiliśmy oczywisty program szukania binarnego (który i tak wydawał się nie mieć zbyt wiele tłuszczyku) superszczupłą wersją, która jest znacznie szybsza (funkcja ta była znana w podziemi programistycznym od wczesnych lat 60. Nauczyłem się jej od Guya Steele’a na początku lat 80., a on sam poznał ją na MIT, gdzie z kolei znano ją od końca lat 60. Vic Vyssotsky użył tego kodu w Bell Labs w 1961 roku; każda instrukcja `if` z pseudokodu została zaimplementowana w trzech instrukcjach systemu IBM 7090).

Narzędzia weryfikacji oprogramowania z rozdziału 4. odegrały zasadniczą rolę w tym zadaniu. Ponieważ z nich korzystaliśmy, możemy być przekonani, że ostateczny program jest poprawny. Kiedy po raz pierwszy zobaczyłem końcową wersję kodu bez jej wyprowadzenia i weryfikacji, patrzyłem na nią, jakby to była magia.

9.4. Reguły

Najważniejsza zasada optymalizacji kodu głosi, że powinno się jej dokonywać rzadko. To szerokie uogólnienie zostało wyjaśnione w następujących punktach:

Rola wydajności. Wiele innych aspektów oprogramowania jest równie ważnych jak wydajność, a może nawet ważniejszych. Don Knuth zauważył, że przedwczesna optymalizacja jest źródłem większości programistycznego zła; może zaważyć na poprawności, funkcjonalności i łatwości obsługi programów. Zachowaj troskę o wydajność do chwili, w której będzie ona mieć znaczenie.

Narzędzia pomiarowe. Kiedy wydajność jest ważna, pierwszym krokiem jest sprofilowanie systemu, aby dowiedzieć się, na co poświęca on najwięcej czasu. Profilowanie programu zwykle pokazuje, że większość czasu jest zużywana w kilku punktach zapalnych, a reszta kodu wykonuje się rzadko (na przykład w podrozdziale 6.1 pojedyncza funkcja jest odpowiedzialna za 98 procent czasu działania programu). Profilowanie prowadzi do krytycznych miejsc; w odniesieniu do pozostałych części kierujemy się mądrą zasadą, że „jeśli nie jest zepsute, nie naprawiaj”. Model kosztów dla prędkości działania, jak ten z dodatku C, może pomóc programiście zrozumieć, dlaczego niektóre operacje i funkcje są kosztowne.

Poziomy projektowania. W rozdziale 6. zobaczyliśmy, że atak na problemy z wydajnością można przypuścić z wielu stron. Przed optymalizacją kodu należy się upewnić, że inne metody nie przyniosą lepszych efektów.

Kiedy przyspieszenia spowalniają. Zastąpienie operatora `%` instrukcją `if` czasami daje dwukrotne przyspieszenie, ale w pozostałych przypadkach nie wpływa na różnicę w czasach działania. Zamiana jednej funkcji na makro dwukrotnie przyspieszyła działanie jednej funkcji, ale za to dziesięć tysięcy razy spowolniła inną funkcję. Po wprowadzeniu „udoskonalenia” niezwykle istotne jest zmierzenie efektu wywieranego przezeń na wzorcowe dane wejściowe. Ze względu na te historie, a także dziesiątki innych, musimy brać pod uwagę ostrzeżenie Jurga Nievergelta kierowane do ulepszaczy kodu: kto tyka bity, może zostać pobity.

W naszej dyskusji rozważaliśmy, czy i kiedy optymalizować kod. Kiedy już się na to zdecydujemy, pozostaje jeszcze kwestia, jak to zrobić. Dodatek D zawiera listę ogólnych zasad, których należy przestrzegać podczas optymalizacji kodu. Wszystkie przykłady, które przeanalizowaliśmy, można wyjaśnić w świetle tych właśnie reguł. Zrobię to teraz, wyróżniając nazwy reguł *pismem pochylonym*.

Program graficzny Van Wyka. Ogólną strategią przyjętą w rozwiązaniu Van Wyka była *eksploatacja najczęstszych przypadków*. W tym przypadku było to *zapisanie w pamięci podręcznej* listy podstawowych typów rekordów.

Problem nr jeden — reszta z dzielenia całkowitego. W rozwiązaniu *wykorzystano tożsamość algebraiczną*, aby zastąpić kosztowną operację dzielenia całkowitego tanim porównaniem.

Problem nr dwa — funkcje, makra i kod inline. *Złożenie hierarchii procedur* przez zastąpienie funkcji makrem przyniosło niemal dwukrotny wzrost prędkości, ale napisanie kodu inline nie sprawiło już żadnej różnicy.

Problem nr trzy — szukania sekwencyjne. Zastosowanie *strażnika* w celu *łączenia testów* dało wzrost prędkości o mniej więcej pięć procent. *Rozwinięcie pętli* przyniosło dodatkowe przyspieszenie o 56 procent.

Problem nr cztery — obliczanie odległości sferycznych. Przechowywanie współrzędnych kartezyjskich razem z szerokościami i długościami geograficznymi stanowi przykład *powiększenia struktury danych*. Użycie tańszych odległości euklidesowych zamiast odległości kątowych to *korzystanie z tożsamości algebraicznej*.

Szukanie binarne. **Łączenie sprawdzania warunków** zmniejszyło liczbę porównań elementów tablicy z dwóch do jednego w pętli wewnętrznej. **Korzystanie z tożsamości algebraicznej** zmieniło reprezentację z górnej i dolnej granicy na granicę dolną i przyrost, a **rozwiniecie pętli** poszerzyło program w celu usunięcia narzutu pętli.

Jak do tej pory pracowaliśmy nad kodem, aby skrócić czas pracy CPU. Kod można optymalizować także w innych celach, takich jak zredukowanie stronicowania albo zwiększenie częstotliwości korzystania z pamięci podręcznej. Prawdopodobnie najczęstszym zastosowaniem optymalizacji kodu oprócz zwiększenia prędkości działania jest zmniejszenie miejsca wymaganego na program. Zagadnieniu temu poświęcony jest następny rozdział.

9.5. Problemy

- (1) Sprofiluj jeden ze swoich programów, a następnie postaraj się skorzystać z metod opisanych w tym rozdziale, aby przyspieszyć działanie programu w jego miejscach zapalnych.
- (2) Witryna tej książki zawiera program w języku C sprofilowany we wstępie. Implementuje on mały podzbiór programów w C++, które zobaczymy w rozdziale 13. Spróbuj sprofilować je w swoim systemie. Jeśli nie masz wyjątkowo wydajnej funkcji `malloc`, Twój system poświęci większość czasu na wykonywanie tej właśnie funkcji. Spróbuj skrócić ten czas, implementując pamięć podręczną dla węzłów, tak jak to zrobił Van Wyk.
- (3) Jakie szczególne właściwości algorytmu „żonglującego” pozwoliły na zastąpienie operatora `%` instrukcją `if` zamiast bardziej kosztowną `while`? Poeksperymentuj, aby określić, kiedy warto zastąpić operator `%` instrukcją `while`.
- (4) Kiedy `n` jest dodatnią liczbą całkowitą nie większą od rozmiaru tablicy, następująca funkcja rekursywna w C zwróci największą wartość w tablicy `x[0..n-1]`:

```
float arrmax(int n)
{
    if (n == 1)
        return x[0];
    else
        return max(x[n-1], arrmax(n - 1));
}
```

Kiedy `max` jest funkcją, znajdzie ona największy element wektora o `n = 10 000` elementach w ciągu kilku sekund. Gdy `max` jest makrem w C

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

algorytm będzie potrzebował 6 sekund na znalezienie wartości maksymalnej dla `n = 27` elementów i 12 sekund na znalezienie wartości maksymalnej dla `n = 28` elementów. Wprowadź dane wejściowe, które sprowokują to okropne zachowanie, i przeanalizuj matematycznie czas działania algorytmu.

- (5) W jaki sposób zachowują się różne algorytmy szukania binarnego, jeśli zostaną zastosowane (wbrew specyfikacji) do nieposortowanych tablic?

- (6) C i C++ udostępniają funkcje klasyfikacji znaków takie jak `isdigit`, `isupper` oraz `islower` w celu określania rodzaju znaków. Jak zaimplementowałbyś te funkcje?
- (7) Mając bardzo długą sekwencję (na przykład miliardy albo tryliony) bajtów, jak można wydajnie policzyć łączną liczbę bitów jedynkowych (to znaczy, ile bitów jest ustawionych w całej sekwencji)?
- (8) Jak można użyć strażników w programie, aby odszukać element tablicy o maksymalnej wartości?
- (9) Ponieważ szukanie sekwencyjne jest prostsze od binarnego, zwykle działa wydajniej w małych tablicach. Z drugiej jednak strony, logarytmiczna liczba porównań w szukaniu binarnym sprawia, że w dużych tablicach jest ono szybsze od szukania liniowego. Punkt zrównania czasów działania zależy od zoptymalizowania każdego z programów. Jak nisko i jak wysoko może znaleźć się ten punkt? Jaki jest on w Twoim komputerze, gdy oba programy są jednako zoptymalizowane?
- (10) D.B. Lomet zauważył, że funkcja mieszająca może rozwiązać problem szukania dla 1000 liczb całkowitych wydajniej niż optymalizacja szukania binarnego. Zaimplementuj szybki program mieszający i porównaj go ze zoptymalizowanym szukaniem binarnym. Jak wypada to porównanie pod względem prędkości i zapotrzebowania na miejsce?
- (11) We wczesnych latach 60. Vic Berecz odkrył, że większość czasu w programach symulacyjnych w firmie Sikorsky Aircraft była poświęcana na liczenie funkcji trygonometrycznych. Dalsze badania wykazały, że funkcje były liczone tylko dla całkowitych wielokrotności pięciu stopni. W jaki sposób udało mu się skrócić czas działania programów?
- (12) Czasami można optymalizować programy, myśląc raczej o matematyce niż o kodzie. Aby obliczyć wielomian

$$y = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0$$

poniższy kod potrzebuje $2n$ mnożeń. Podaj szybszą funkcję.

```
y = a[0]
xi = 1
for i = [1, n]
    xi = x * xi
    y = y + a[i]*xi
```

9.6. Lektury uzupełniające

W podrozdziale 3.8 wspomniano *Kod doskonały* Steve'a McConnella. W rozdziale 25. autor opisuje strategie optymalizacji kodu; rozdział ten zawiera ogólne spojrzenie na wydajność i szczegółowe opisy metod optymalizacji kodu. W rozdziale 26. znajduje się wspaniała kolekcja reguł optymalizacji kodu.

W dodatku D do tej książki zaprezentowano pokrewne reguły optymalizacji kodu wraz z opisem, w jaki sposób zostały one zastosowane w niniejszej publikacji.

Skorowidz

A

algorytm dziel i zwyciężaj, 89–91
algorytm kwadratowy, 88–89
algorytm skanujący, 91
algorytmy, 15, 21–29, 43–44, 87–96, 139–142, 169–170, 193–197
 algorytm 40-przebiegowy, 15
 algorytm dziel i zwyciężaj, 89–91
 algorytm kwadratowy, 88–89
 algorytm skanujący, 91
 algorytm sortujący, 169–170
 drzewa szukania binarnego, 195
 euklidesowy algorytm addytywny, 197
 kolejki priorytetowe, 196
 mieszanie, 195
 problem A, 21, 23
 problem B, 21, 23
 problem C, 22, 25, 29
 problemy, 21–22, 27–28
 prosty algorytm, 87–88
 przestrzeń projektowania, 139–142
 reguły, 26–27
 sortowanie, 25–26
 sortowanie przez kopcowanie, 194
 sortowanie przez scalanie, 194
 sortowanie przez wstawianie, 193
 sortowanie szybkie, 194
 sygnatury, 26
 szukanie binarne, 22, 43–44, 195
 szukanie sekwencyjne, 22, 195
 techniki projektowania, 87–96
 wybór, 196
arkusze kalkulacyjne, 38
asercja, 59–60

B

bazy danych, 38
bigrand, 138, 205
binarysearch, 57
bisekcja, 23
break, 45
bsearch, 186

C

case, 48
comlen, 179, 180

D

debugowanie, 66–67
drzewa szukania binarnego, 150–152, 195

E

extractmin, 166, 168

F

for, 59, 176
Fortrana tablice, 112
funkcje
 bigrand, 138, 205
 binarysearch, 57
 bsearch, 186
 comlen, 179, 180
 extractmin, 166, 168
 incword, 177

funkcje

inline, 99, 126
 insert, 37, 146, 150, 153, 154, 166, 168
 IntSet, 146
 main, 57, 177, 203
 malloc, 81, 185
 max, 99
 mustbe, 45
 pstrcmp, 180
 qsort, 129, 180, 186
 rand, 205
 randint, 131, 138
 report, 146, 148, 149, 153, 154
 rinsert, 150, 154
 scanf, 183
 search, 37
 siftdown, 165, 168, 169, 171
 siftup, 163, 164, 167, 169, 171
 size, 146, 148
 sizeof, 201
 sortcmp, 184
 strcmp, 180, 186
 strdup, 178
 swap, 126, 132
 uncheckall, 36
 warunek końcowy, 49
 warunek wstępny, 49
 weryfikacja, 49
 wordncmp, 186

G

generator korespondencji seryjnej, 34

H

hipertekst, 37

I

if, 59, 104
 implementacja programu anagramowego, 29
 incword, 177
 inline, 99, 126
 insert, 37, 146, 150, 153, 154, 166, 168
 IntSet, 146
 IntSetImp, 146

J

języki dziedziczne, 39

K

kopce, 161–173
 algorytm sortujący, 169–170
 kolejki priorytetowe, 161, 166–169
 kolejność, 171
 krytyczne funkcje, 163–166
 kształt, 162–163, 171
 problemy, 171–173
 reguły, 171
 sortowanie, 161
 specyfikacja, 166
 struktura danych, 161–163
 uporządkowanie, 161

L

Little'a prawo, 83–84

Ł

łańcuchy znaków, 175–187
 frazy, 178–181
 generowanie tekstu, 181–185
 problemy, 186–187
 reguły, 185–186
 słowa, 175–178

M

main, 57, 177, 203
 malloc, 81, 97, 185
 max, 99
 mustbe, 45

N

new, 81, 201

O

obliczenia pobieżne, 77–85
 czynniki bezpieczeństwa, 82–83
 dwie odpowiedzi są lepsze niż jedna, 78
 praktyczne zasady, 79

praktyka, 80
 prawo Little'a, 83–84
 problemy, 84–85
 reguły, 84
 szacowanie wydajności, 80–82
 szybkie sprawdzanie, 78
 optymalizacja kodu, 97–108, 207–212
 funkcje, makra i kod inline, 99–100
 obliczanie odległości sferycznych, 101–102
 problemy, 107–108
 profilowanie programu, 97
 reguły, 105–107, 207–212
 reguły „czas kosztem miejsca”, 208
 reguły logiczne, 209
 reguły „miejsce kosztem czasu”, 207
 reguły pętli, 208
 reguły procedur, 210
 reguły wyrażań, 211
 reszta z dzielenia całkowitego, 98–99
 strażnik, 100
 szukania sekwencyjne, 100–101
 szukanie binarne, 102–105
 zapisywania w pamięci podręcznej, 98
 oszczędzanie miejsca, 109–120
 problem pogładowy, 110–113
 problemy, 119–120
 reguły, 118–119
 tablice Fortrana, 112
 techniki zmniejszające wielkość danych, 113–116
 techniki zmniejszające wielkość kodu, 116–117

P

pary nazwa-wartość, 38
 poziomy projektowania, 71, 73–74
 algorytmy i struktury danych, 74
 definicja problemu, 74
 oprogramowanie systemowe, 74
 optymalizacja kodu, 74
 sprzęt, 74
 struktura systemu, 74
 prawo Little'a, 83–84
 print, 62
 printf, 58, 180
 problem próby, 137–142
 rozwiązanie, 138–139
 program kwestionariuszowy, 31–33
 programowanie listów seryjnych, 33–35

generator korespondencji seryjnej, 34
 schemat listu seryjnego, 34
 programy, 31–41, 43–53, 55–65, 137–144, 145–156
 arkusze kalkulacyjne, 38
 bazy danych, 38
 binarne drzewa poszukiwań, 150–152
 hipertekst, 37
 implementacja w C, 55–65
 języki dziedzinowe, 39
 pary nazwa-wartość, 38
 pisanie programu, 44–46
 pomiar prędkości, 61–62
 problem próby, 137–142
 problemy, 40–41, 51–53, 64–65, 143–144, 156
 program kwestionariuszowy, 31–33
 programowanie listów seryjnych, 33–35
 przestrzeń projektowania, 139–142
 reguły, 39–40, 49–50, 63–64, 142–143, 155
 struktury dla liczb całkowitych, 153–154
 struktury liniowe, 147–150
 strukturyzacja danych, 37
 szukanie, 145–156
 tablica przykładów, 35–37
 testowanie zautomatyzowane, 60–61
 weryfikacja kodu, 46–48
 pstrcmp, 180

Q

qsort, 129, 180, 186

R

rand, 205
 randint, 131, 138
 reguły, 17–18, 26–27, 39–40, 49–50, 63–64, 75, 84, 93–94, 105–107, 118–119, 132–133, 142–143, 155, 171, 185–186, 207–212
 abstrakcja, 171
 abstrakcja proceduralna, 171
 abstrakcyjne typy danych, 171
 algorytmy dziel i zwyciężaj, 94
 algorytmy skanujące, 94
 algorytmy wieloprzebiegowe, 17
 asercje, 49
 biblioteki czy komponenty niestandardowe?, 186
 bitmapowa struktura danych, 17
 debugowanie, 64

reguły

- definicja problemu, 27
- dokonaj enkapsulacji złożonych struktur, 40
- drzewa zrównoważone, 186
- eksploatacja najczęstszych przypadków, 106
- eliminacja wspólnych podwyrażeń, 212
- eliminacja zmiennych logicznych, 210
- etapy projektowania programu, 18
- funkcje, 49
- inicjalizacja w czasie kompilacji, 211
- interpretery, 208
- istotność objętości, 155
- iteracyjne struktury kontrolne, 49
- kiedy potrzebujesz przyspieszenia, 75
- kiedy przyspieszenia spowalniają, 106
- kodowanie, 64
- kompromis prędkość kosztem pamięci, 17
- kompromisy, 118
- korzystanie z tożsamości algebraicznej, 106–107
- korzystanie z zaawansowanych narzędzi, 40
- koszt pamięci, 118
- łączenie sprawdzania warunków, 107
- łączenie testów, 209
- mierzenie objętości, 118
- mieszanie, 185
- narzędzia pomiarowe, 106
- nie pisz dużego programu, jeśli wystarczy mały, 39
- niech dane strukturyzują program, 40
- ograniczenia dolne, 94
- paralelizm, 211
- perspektywa osoby rozwiązującej problem, 27
- poprawność, 171
- powiększanie struktur danych, 207
- powiększanie struktury danych, 106
- poziomy projektowania, 106
- pracuj w środowisku, 118
- prędkość, 64
- prosta projektu, 18
- przenoszenie kodu poza pętlę, 208
- „punkty zapalne” pamięci, 118
- retrospektywa, 143
- rola bibliotek, 155
- rola wydajności, 106
- rozwiniecie pętli, 106, 209
- rusztowanie, 63
- scalanie pętli, 209
- sekwencyjne struktury kontrolne, 49
- składanie hierarchii funkcji, 210

skręcanie obliczeń funkcji

- monotonicznych, 209
- sortowanie, 26
- stosowanie pamięci podręcznej, 208
- struktury danych dedykowane łańcuchom, 185
- struktury kontrolne wyboru, 49
- sygnatury, 26
- szukanie binarne, 26
- tablice sufiksowe, 186
- techniki optymalizacji kodu, 155
- testowanie, 64
- transformacje funkcji rekursywnych, 211
- upakowanie, 208
- usuwanie rozgałęzień bezwarunkowych, 209
- używanie właściwych narzędzi, 119
- wartości skumulowane, 94
- wartościowanie leniwe, 208
- właściwy problem, 17
- współprogramy, 210
- wstępne obliczanie funkcji logicznych, 210
- wstępnie przetwórz informacje na struktury danych, 94
- wydajność, 171
- wykorzystanie paralelizmu słowa, 212
- wykorzystanie tożsamości algebraicznych, 209–211
- wykorzystanie wspólnych przypadków, 210
- wyspecyfikuj abstrakcję problemu, 142
- zaimplementuj jedno rozwiązanie, 142
- zamień powtarzający się kod na tablice, 40
- zapamiętuj stan, 94
- zapisanie w pamięci podręcznej, 106
- zapisywanie wstępnie uzyskanych wyników, 207
- zbadaj przestrzeń projektu, 142
- zestawianie obliczeń w pary, 212
- złożenie hierarchii procedur, 106
- zmiana kolejności testów, 210
- zrozum zaobserwowany problem, 142
- rekursja, 150
- report, 146, 148, 149, 153, 154
- rinsert, 150, 154
- rotldist, 99

S

- scanf, 183
- schemat listu seryjnego, 34
- search, 37

siftdown, 165, 168, 169, 171
siftup, 163, 164, 167, 169, 171
size, 146, 148
sizeof, 81, 201
sortcmp, 184
sortowanie, 14–19, 25–26, 125–134, 169–170, 193–194
 algorytm sortujący, 169–170
 postać kanoniczna, 26
 problemy, 133–134
 reguły, 132–133
 sortowanie bitmapowe, 194
 sortowanie pliku dyskowego, 14–19
 sortowanie pozycyjne, 194
 sortowanie przez kopcowanie, 194
 sortowanie przez scalanie, 194
 sortowanie przez wstawianie, 125–127, 193
 sortowanie szybkie, 127, 132, 194
 sortowanie wieloprzebiegowe, 194
 sygnatury, 26
 wymagania względem danych
 wyjściowych, 193
 zestawianie takich samych elementów, 193
sortowanie pliku dyskowego, 14–19
 algorytm 40-przebiegowy, 15
 ograniczenia, 14
 problemy, 18–19
 reguły, 17–18
 szkic implementacji, 16
 wejście, 14
 wektor bitowy, 16
 wyjście, 14
strażnik, 100, 147, 209
 maxval, 147
strcmp, 180, 186
strdup, 178
strukturyzacja danych, 37
swap, 126, 132
switch, 62
sygnatury, 26
szukanie, 22–23, 26, 43–48, 55–63, 100–101, 102–105 145–156, 157–159, 194–196
 drzewa szukania binarnego, 150–152, 195
 indeksowanie po kluczu, 195
 interfejs, 145–147
 mieszanie, 195
 problemy, 156
 struktury dla liczb całkowitych, 153–154
 struktury liniowe, 147–150

 szukanie binarne, 22–23, 26, 43–48, 55–63, 102–105, 195
 szukanie sekwencyjne, 22, 100–101, 195
 zastosowania, 195
szukanie binarne, 22–23, 26, 43–48, 55–63, 102–105, 195
 asercja, 59–60
 bisekcja, 23
 implementacja w C, 55–63
 pisanie programu, 44–46
 próbkiowanie, 22
 weryfikacja kodu, 46–48
szukanie sekwencyjne, 22, 100–101, 195

T

tablica przykładów, 35–37
 analiza słów, 36
 funkcje daty, 36
 komunikaty o błędach, 36
 menu, 35
tablice Fortrana, 112
techniki projektowania algorytmów, 87–96
 algorytm dziel i zwyciężaj, 89–91
 algorytm skanujący, 91
 dwa algorytmy kwadratowe, 88–89
 problemy, 94–96
 prosty algorytm, 87–88
 reguły, 93–94
techniki zmniejszające wielkość danych, 113–116
 dynamiczna alokacja, 115
 indeksowanie kluczem, 114
 kompresja danych, 114–115
 nie przechowuj, obliczaj, 113–114
 odśmiecianie, 115
 polityka alokacji, 115
 polityka rekordów o zmiennej długości, 115
 rzadkie struktury danych, 114
 współdzielenie pamięci, 115
techniki zmniejszające wielkość kodu, 116–117
 definiowanie funkcji, 117
 interpretery, 117
 tłumaczenie na język maszynowy, 117

U

uncheckedall, 36

W

wektor bitowy, 16
weryfikacja programów, 49–50
 reguły, 49–50
while, 176
wordncmp, 186
wydajność, 71–73, 75–76, 77–85, 87–96, 97–108,
 109–120, 201–205
 algorytmy i struktury danych, 72
 obliczenia pobieżne, 77–85
 optymalizacja algorytmu, 72
 optymalizacja kodu, 72, 97–108
 oszczędzanie miejsca, 109–120
 problemy, 75–76
 reguły, 75
 reorganizacja struktury danych, 72
 sprzęt, 73
 szacowanie wydajności, 201–205
 techniki projektowania algorytmów, 87–96

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**

PERŁKI PROGRAMOWANIA KANON INFORMATYKI

Każdy programista w swojej karierze spotyka się z ciekawymi, intrygującymi i czasami skomplikowanymi problemami. A te potrafią drażnić, pobudzać ciekawość i zmuszać do ciągłego zastanawiania się nad nimi przez wiele dni i tygodni o każdej porze dnia i nocy. Aż nagle... W głowie pojawia się rozwiązanie – najlepsze z możliwych, eleganckie, wydajne i proste. To rozwiązanie jest właśnie perłką!

Ta książka jest zbiorem takich perł. Znajdziesz w niej dziesiątki ciekawych problemów i jeszcze ciekawszych rozwiązań. Autor omawia w niej istotę algorytmów, sposoby na poprawę wydajności programu oraz najlepsze techniki sortowania i kompresji danych. Twoją ciekawość wzbudzi z pewnością katalog algorytmów, w którym być może znajdziesz optymalny algorytm rozwiązujący Twoje największe problemy. To jedna z najbardziej popularnych i wpływowych publikacji informatycznych – jej zaktualizowana wersja z pewnością utrwali tę pozycję. Ta książka po prostu musi się znaleźć na Twojej półce!

Najbardziej wpływowa książka w świecie informatyki w odnowionej wersji!

- Określanie problemu
- Algorytmy
- Sposoby pisania poprawnych programów
- Droga od pseudokodu do programu
- Testowanie programu
- Szacowanie i mierzenie wydajności
- Techniki projektowania algorytmów
- Optymalizacja kodu
- Kompresja kodu oraz danych
- Szybkie sortowanie oraz wydajne wyszukiwanie
- Kopce
- Katalog algorytmów

helion.pl
księgarnia
internetowa

Nr katalogowy: 6796



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje
☛ <http://helion.pl/promocje>
Książki najchętniej czytane
☛ <http://helion.pl/bestsellery>
Zamów informacje o nowościach
☛ <http://helion.pl/novosci>

Helion SA
ul. Kosciuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>



ISBN 978-83-246-3481-1



Cena 49,00 zł

Informatyka w najlepszym wydaniu