helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Piękny kod. Tajemnice mistrzów programowania

Autor: Andy Oram, Greg Wilson
Tłumaczenie: Łukasz Piwko, Marcin Rogóż
ISBN: 978-83-246-1408-0
Tytuł oryginału: Beautiful Code: Leading Programmers Explain How They Think
Format: 168x237, stron: 564

Poznaj techniki pracy guru programowania!

- Jak tworzyć czytelny i pozbawiony błędów kod?
- W jaki sposób projektować architekturę systemów?
- Jak zbudować uniwersalne interfejsy użytkownika?

Wbrew pozorom programowanie to nie tylko nauka ścisła, to także sztuka! Trudna sztuka! Napisanie kodu poprawnie działającego czy kodu spełniającego oczekiwania użytkowników programu to niewątpliwie wyzwanie! Wymaga bowiem doskonałego zaplanowania architektury, skutecznej optymalizacji kodu źródłowego oraz umiejętności przewidywania potencjalnych problemów i ich odpowiednio wczesnej eliminacji. Właśnie w tej książce prawdziwi mistrzowie programowania podzielą się z Tobą swoimi doświadczeniami, przemyśleniami i spostrzeżeniami dotyczącymi tworzenia profesjonalnych rozwiązań. Znajdziesz tu wiele praktycznych porad dotyczących pisania kodu, rozwiązywania problemów programistycznych, projektowania architektury, tworzenia interfejsów użytkownika i pracy w zespole projektowym. Dowiesz się, kiedy należy postępować dokładnie według wskazań metodologii, a kiedy „pójście na skróty" może okazać się najlepszym rozwiązaniem. Poznasz sposób myślenia i zasady pracy najlepszych programistów świata, dzięki czemu użytkownikom Twoich aplikacji zapewnisz maksymalny komfort.

- Korzystanie z wyrażeń regularnych
- Dobór odpowiedniego poziomu abstrakcji
- Ocena jakości kodu źródłowego
- Testowanie
- Techniki analizy składni
- Zabezpieczanie komunikacji sieciowej
- Dostosowywanie architektury systemu do architektury komputerów
- Praca zespołowa
- Projektowanie systemów w oparciu o komponenty OpenSource
- Usuwanie błędów
- Ułatwianie pracy osobom niepełnosprawnym

Dołącz do grona mistrzów programowania!

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

SPIS TREŚCI

Słowo wstępne **13**

Wstęp **15**

1. Wyrażenia regularne **19**
Programowanie w praktyce 20
Implementacja 21
Omówienie 22
Alternatywy 24
Rozszerzanie 25
Podsumowanie 27

2. Edytor delty w Subversion — interfejs jako ontologia **29**
Kontrola wersji i transformacja drzewa 30
Prezentacja różnic pomiędzy drzewami 34
Interfejs edytora delty 35
Ale czy to jest sztuka? 40
Abstrakcja jako sport widowiskowy 43
Wnioski 45

3. Najpiękniejszy kod, którego nigdy nie napisałem **47**
Najpiękniejszy kod, jaki kiedykolwiek napisałem 47
Coraz więcej za pomocą coraz mniejszych środków 49
Perspektywa 54
Co to jest pisanie 57
Zakończenie 57
Podziękowania 59

4. Wyszukiwanie **61**
Na czas 61
Problem — dane z pamiętnika sieciowego 62
Problem — kto zażądał, czego i kiedy 70
Wyszukiwanie na dużą skalę 75
Podsumowanie 77

5. Poprawny, piękny, szybki (w takiej kolejności) — lekcje z projektowania weryfikatorów XML **79**
Znaczenie walidacji XML 79
Problem 80
Wersja 1. Naiwna implementacja 82
Wersja 2. Imitacja gramatyki BNF O(N) 83
Wersja 3. Pierwsza optymalizacja O(log N) 84
Wersja 4. Druga optymalizacja — nie sprawdzaj dwa razy 85
Wersja 5. Trzecia optymalizacja O(1) 87
Wersja 6. Czwarta optymalizacja — buforowanie 91
Morał 93

6. Framework for Integrated Test — piękno poprzez delikatność **95**
Acceptance Testing Framework w trzech klasach 96
Wyzwanie zaprojektowania środowiska 98
Otwarte środowisko 99
Jak prosty może być parser HTML 100
Podsumowanie 103

7. Piękne testy **105**
To niesforne wyszukiwanie binarne 106
Wstęp do JUnit 109
Rozprawić się z wyszukiwaniem binarnym 111
Podsumowanie 122

8. Generowanie w locie kodu do przetwarzania obrazów **125**

9. Kolejność wykonywania operatorów **147**
JavaScript 148
Tablica symboli 149
Tokeny 150
Kolejność 151
Wyrażenia 152
Operatory wrostkowe 152
Operatory przedrostkowe 154
Operatory przypisania 155
Stałe 155
Zakres 156
Instrukcje 157
Funkcje 160
Literały tablicowe i obiektowe 161
Rzeczy do zrobienia i przemyślenia 162

10. Poszukiwanie szybszych metod zliczania bitów w stanie wysokim **163**
Podstawowe metody 164
Dziel i zwyciężaj 165
Inne metody 167

Suma i różnica liczb ustawionych bitów w dwóch słowach 169
Porównywanie liczby ustawionych bitów w dwóch słowach 169
Zliczanie jedynek w tablicy 170
Zastosowania 175

11. Bezpieczna komunikacja — technologia wolności 177
Początki 178
Rozwikłać tajemnicę bezpiecznego przesyłania wiadomości 180
Klucz to użyteczność 181
Podstawa 184
Zestaw testów 188
Działający prototyp 189
Oczyść, podłącz i używaj 190
Hakowanie w Himalajach 194
Niewidoczne ruchy ręką 199
Prędkość ma znaczenie 201
Prywatność komunikacji dla praw jednostki 202
Hakowanie cywilizacji 203

12. Hodowanie pięknego kodu w języku BioPerl 205
BioPerl i moduł Bio::Graphics 206
Proces projektowania modułu Bio::Design 210
Rozszerzanie modułu Bio::Graphics 228
Wnioski i lekcje 232

13. Projekt programu Gene Sorter 235
Interfejs użytkownika programu Gene Sorter 236
Podtrzymywanie dialogu z użytkownikiem przez internet 237
Nieco polimorfizmu 239
Filtrowanie w celu znalezienia odpowiedniego genu 242
Ogólna teoria pięknego kodu 243
Podsumowanie 246

14. Jak elegancki kod ewoluuje wraz ze sprzętem — przypadek eliminacji Gaussa 247
Wpływ architektury komputerów na algorytmy macierzowe 248
Metoda dekompozycyjna 250
Prosta wersja 251
Podprocedura DGEFA biblioteki LINPACK 252
Procedura LAPACK DGETRF 255
Rekursywna dekompozycja LU 257
Procedura ScaLAPACK PDGETRF 260
Wielowątkowość w systemach wielordzeniowych 265
Słowo na temat analizy błędów i liczby operacji 267
Przyszłe kierunki badań 268
Literatura zalecana 269

15. Długoterminowe korzyści z pięknego projektu 271
Moje wyobrażenie o pięknym kodzie 271
Wprowadzenie do biblioteki CERN 272
Zewnętrzne piękno 273
Piękno wewnętrzne 278
Podsumowanie 284

16. Model sterowników jądra systemu Linux — korzyści płynące ze współpracy 285
Skromne początki 286
Redukcja do jeszcze mniejszych rozmiarów 290
Skalowanie do tysięcy urządzeń 293
Małe, luźno połączone obiekty 294

17. Inny poziom pośredniości 297
Od kodu do wskaźników 297
Od argumentów funkcji do wskaźników argumentów 300
Od systemów plików do warstw systemów plików 303
Od kodu do języka konkretnej domeny 305
Multipleksacja i demultipleksacja 307
Na zawsze warstwy? 308

18. Implementacja słownika w Pythonie — być wszystkim dla wszystkich 311
Wewnątrz słownika 313
Warunki specjalne 314
Kolizje 316
Zmiana rozmiaru 317
Iteracje i zmiany dynamiczne 318
Podsumowanie 319
Podziękowania 319

19. Wielowymiarowe iteratory w NumPy 321
Kluczowe wyzwania w operacjach na N-wymiarowych tablicach 322
Modele pamięci dla tablicy N-wymiarowej 323
Początki iteratora NumPy 324
Interfejs iteratora 331
Wykorzystanie iteratora 332
Podsumowanie 336

20. System korporacyjny o wysokim stopniu niezawodności dla misji Mars Rover NASA 337
Misja i Collaborative Information Portal 338
Wymagania misji 339
Architektura systemu 340
Studium przypadku — usługa strumieniowa 343
Niezawodność 346
Solidność 353
Podsumowanie 355

21. ERP5 — projektowanie maksymalnej giętkości **357**
Ogólne cele ERP 358
ERP5 358
Podstawowa platforma Zope 360
Założenia ERP5 Project 364
Pisanie kodu dla ERP5 Project 365
Podsumowanie 368

22. Łyżka dziegciu **371**

23. Programowanie rozproszone z zastosowaniem MapReduce **389**
Motywujący przykład 389
Model programistyczny MapReduce 392
Inne przykłady MapReduce 393
Implementacja rozproszonego MapReduce 394
Rozszerzenia modelu 398
Wnioski 399
Literatura zalecana 400
Podziękowania 400
Dodatek: przykład algorytmu zliczającego słowa 400

24. Piękna współbieżność **403**
Prosty przykład: konta bankowe 404
Pamięć transakcyjna STM 406
Problem Świętego Mikołaja 414
Refleksje na temat Haskella 422
Wnioski 423
Podziękowania 424

25. Abstrakcja składniowa — rozszerzenie syntax-case **425**
Krótkie wprowadzenie do syntax-case 429
Algorytm rozwijania 431
Przykład 443
Wnioski 445

26. Architektura oszczędzająca nakłady — obiektowy framework dla oprogramowania sieciowego **447**
Przykładowa aplikacja — usługa rejestrowania 449
Zorientowany obiektowo projekt frameworku serwera rejestrowania 451
Implementacja sekwencyjnych serwerów rejestrowania 457
Implementacja współbieżnych serwerów rejestrowania 461
Wnioski 467

27. Integracja partnerów biznesowych z wykorzystaniem architektury REST **469**
Tło projektu 470
Udostępnianie usług klientom zewnętrznym 470

Przekazywanie usługi za pomocą wzorca fabryki 473
Wymiana danych z użyciem protokołów e-biznesowych 475
Wnioski 480

28. Piękne debugowanie 481
Debugowanie debugera 482
Systematyczny proces 483
Szukany problem 485
Automatyczne wyszukiwanie przyczyny awarii 486
Debugowanie delta 488
Minimalizacja wejścia 490
Polowanie na usterkę 490
Problem prototypu 493
Wnioski 493
Podziękowania 494
Literatura zalecana 494

29. Traktując kod jako esej 495

30. Gdy ze światem łączy cię tylko przycisk 501
Podstawowy model projektu 502
Interfejs wejściowy 505
Wydajność interfejsu użytkownika 518
Pobieranie 518
Przyszłe kierunki rozwoju 519

31. Emacspeak — kompletne dźwiękowe środowisko pracy 521
Tworzenie wyjścia mówionego 522
Włączanie mowy w Emacsie 523
Bezbolesny dostęp do informacji online 534
Podsumowanie 541
Podziękowania 544

32. Kod w ruchu 545
O byciu „podręcznikowym” 546
Podobne wygląda podobnie 547
Niebezpieczeństwa wcięć 548
Poruszanie się po kodzie 549
Wykorzystywane przez nas narzędzia 550
Burzliwa przeszłość DiffMerge 552
Wnioski 554
Podziękowania 554
Literatura zalecana 554

33. Pisanie programów dla Księgi **557**
Niekrólewska droga 558
Ostrzeżenie dla nawiasofobów 558
Trzy w rzędzie 559
Śliskie nachylenie 561
Nierówność trójkąta 563
Meandrowanie 565
„No przecież!", znaczy się „Aha!" 566
Wnioski 567
Zalecana literatura 568

Posłowie **571**

Autorzy **573**

Skorowidz **583**

ROZDZIAŁ 3.

Najpiękniejszy kod, którego nigdy nie napisałem

Jon Bentley

Kiedyś słyszałem, że pewien mistrz programowania dawał taką oto pochwałę: „On dodaje funkcje poprzez usuwanie kodu". Antoine Saint-Exupéry, francuski pisarz i lotnik, wyraził tę myśl bardziej ogólnie: „Projektant może uznać, że osiągnął perfekcję, nie wtedy, kiedy nie pozostało już nic do dodania, ale wtedy, gdy nie można już nic odjąć". W oprogramowaniu najpiękniejszego kodu, najpiękniejszych funkcji i najpiękniejszych programów czasami w ogóle nie ma.

Oczywiście trudno dyskutować o rzeczach, których nie ma. Ten rozdział jest próbą wykonania tego przytłaczającego zadania poprzez zaprezentowanie nowatorskiej analizy czasu pracy klasycznego programu Quicksort. Pierwszy podrozdział zawiera ogólny opis programu z osobistego punktu widzenia. Następny — to już treść właściwa tego rozdziału. Zaczniemy od dodania jednego licznika do programu, a następnie będziemy manipulować kodem, żeby stawał się coraz mniejszy i potężniejszy, aż tylko kilka wierszy kodu w pełni będzie pokrywać jego średni czas działania. Trzeci podrozdział podsumowuje techniki i zawiera niezwykle zwięzłą analizę kosztów binarnych drzew poszukiwań. Wskazówki znajdujące się w dwóch ostatnich podrozdziałach, oparte na spostrzeżeniach zawartych w tym tekście, pomogą nam pisać bardziej eleganckie programy.

Najpiękniejszy kod, jaki kiedykolwiek napisałem

Kiedy Greg Wilson przedstawił mi pomysł na tę książkę, zadałem sobie pytanie, jaki był najpiękniejszy kod, który napisałem. Po prawie całym dniu kołatania się tego pytania w mojej głowie zdałem sobie sprawę, że ogólna odpowiedź jest niezwykle prosta: Quicksort. Jednak w zależności od tego, jak precyzyjnie sformułuje się to pytanie, można odpowiedzieć na nie na trzy sposoby.

Tematem mojej rozprawy naukowej były algorytmy typu „dziel i zwyciężaj”. Dzięki niej odkryłem, że algorytm Quicksort napisany przez programistę o nazwisku C. A. R. Hoare (*Quicksort*, „Computer Journal” nr 5) jest niezaprzeczalnie dziadkiem ich wszystkich. Jest to piękny algorytm rozwiązujący podstawowy problem, który można zaimplementować w eleganckim kodzie. Zawsze go uwielbiałem, ale trzymałem się z dala od jego najgłębiej zagnieżdżonej pętli. Kiedyś spędziłem dwa dni na debugowaniu programu opartego na niej i całymi latami kopiowałem skrupulatnie kod za każdym razem, kiedy musiałem wykonać podobne zadanie. Rozwiązywał moje problemy, ale nigdy tak *naprawdę* go nie rozumiałem.

W końcu nauczyłem się od Nico Lomuto eleganckiej metody dzielenia i nareszcie mogłem napisać program Quicksort, który byłby dla mnie zrozumiały, a nawet umiałbym udowodnić, że jest poprawny. Spostrzeżenie Williama Strunka Jr., że „piszący szybko piszą zwięźle”, ma zastosowanie zarówno do kodu, jak i języka angielskiego. W związku z tym, idąc za jego radą, „pomijałem zbędne słowa” (*The Elements of Style*). Udało mi się zredukować 40 wierszy kodu do równo 12. A więc jeśli pytanie brzmi: „Jaki jest najpiękniejszy mały fragment kodu, jaki w życiu napisałem?”, moja odpowiedź to: Quicksort z mojej książki pod tytułem *Perełki oprogramowania*[1]. Ta funkcja Quicksort, napisana w języku C, została przedstawiona na listingu 3.1. W następnym podrozdziale zajmiemy się dalszym dostrajaniem i badaniem tego kodu.

LISTING 3.1. Funkcja Quicksort

```
void quicksort(int l, int u)
{   int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

Kod ten sortuje globalną tablicę `x[n]`, kiedy jest wywoływany z argumentami `quicksort(0, n-1)`. Oba argumenty tej funkcji są indeksami podtablicy, która ma być posortowana. `l` oznacza dolną granicę (ang. *lower*), a `u` — górną (ang. *upper*). Wywołanie funkcji `swap(i,j)` powoduje zamianę zawartości elementów `x[i]` i `x[j]`. Pierwsza funkcja `swap` losowo wybiera element podziału, który w taki sam sposób jest wybierany spomiędzy `l` i `u`.

Książka *Perełki oprogramowania* zawiera szczegółowy opis i dowód poprawności funkcji `quicksort`. Zakładam, że Czytelnik zna algorytm Quicksort na poziomie tamtego opisu i najbardziej podstawowych książek o algorytmach.

Jeśli zmienimy pytanie na: „Jaki jest najpiękniejszy powszechnie używany fragment kodu, który napisałeś?”, moja odpowiedź ponownie będzie brzmieć Quicksort. W artykule napisanym razem

[1] Jon Bentley, *Perełki oprogramowania*, wyd. 2, Wydawnictwa Naukowo-Techniczne, Warszawa 2001 — *przyp. red.*

z M. D. McIlroyem[2] omawiamy poważny błąd związany z wydajnością w nieco sędziwej już funkcji systemu Unix — `qsort`. Wzięliśmy się za pisanie nowej funkcji `sort` dla biblioteki języka C, biorąc pod uwagę wiele różnych algorytmów do wykorzystania, w tym *Merge Sort* i *Heap Sort*. Po porównaniu kilku możliwości implementacji zdecydowaliśmy się na wersję z algorytmem Quicksort. We wspomnianym artykule wyjaśniamy, w jaki sposób napisaliśmy nową funkcję, która była bardziej przejrzysta, szybsza i solidniejsza niż jej konkurentki — po części z racji swoich niewielkich rozmiarów. Mądra rada Gordona Bella okazała się słuszna: „Najtańsze, najszybsze i najbardziej niezawodne komponenty systemu komputerowego to te, których nie ma". Funkcja ta jest już powszechnie używana od ponad dziesięciu lat i nie zgłoszono jeszcze żadnych błędów.

Biorąc pod uwagę korzyści płynące ze zmniejszania objętości kodu, zadałem sobie w końcu trzeci wariant pytania zamieszczonego na początku tego rozdziału: „Jaki jest najpiękniejszy fragment kodu, którego *nigdy* nie napisałem?". Jak udało mi się osiągnąć bardzo dużo za pomocą tak małych środków? Odpowiedź i tym razem jest związana z Quicksort, a konkretnie z analizą jego wydajności. O tym opowiadam w kolejnym podrozdziale.

Coraz więcej za pomocą coraz mniejszych środków

Quicksort to bardzo elegancki algorytm, który nadaje się do wykonywania wnikliwych analiz. Około roku 1980 odbyłem wspaniałą rozmowę z Tonym Hoarem na temat historii jego algorytmu. Powiedział mi, że kiedy go opracował, wydawało mu się, iż jest on zbyt prosty do opublikowania. Napisał więc tylko swój klasyczny artykuł *Quicksort*, kiedy udało mu się przeanalizować jego oczekiwany czas wykonywania.

Łatwo się zorientować, że posortowanie tablicy zawierającej n elementów algorytmowi Quicksort może w najgorszym przypadku zająć około n^2 czasu. W najlepszym natomiast przypadku wybiera on wartość średnią jako element dzielący, dzięki czemu sortuje tablicę za pomocą około $n \times \lg(n)$ porównań. A więc ilu średnio porównań potrzebuje w przypadku losowej tablicy n różnych wartości?

Analiza tego problemu dokonana przez Hoare'a jest piękna, ale niestety wykraczająca poza wiedzę matematyczną wielu programistów. Kiedy uczyłem zasady działania algorytmu Quicksort studentów, martwiło mnie, że wielu z nich nie mogło zrozumieć dowodu, nawet mimo mojego szczerego wysiłku. Spróbujemy teraz podejść do tego zagadnienia w eksperymentalny sposób. Zaczniemy od programu Hoare'a i stopniowo dojdziemy do analizy zbliżonej do jego własnej.

Naszym zadaniem jest zmodyfikować kod z listingu 3.1 przedstawiającego randomizujący kod Quicksort, aby drogą analizy sprawdzał średnią liczbę porównań potrzebnych do posortowania tablicy zawierającej unikatowe elementy. Spróbujemy też uzyskać jak najwięcej przy użyciu jak najmniejszej ilości kodu, czasu i miejsca.

Aby określić średnią liczbę porównań, najpierw rozszerzymy funkcjonalność programu o możliwość ich zliczania. W tym celu inkrementujemy zmienną `comps` przed porównaniem w wewnętrznej pętli (listing 3.2).

---

[2] J. Bentley, M. D. McIlroy, *Engineering a sort function*, „Software-Practice and Experience", Vol. 23, No. 11 — *przyp. red.*

LISTING 3.2. Wewnętrzna pętla algorytmu Quicksort przystosowana do zliczania porównań

```
for (i = l+1; i <= u; i++) {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

Jeśli uruchomimy program tylko dla jednego *n*, dowiemy się, ile porównań to jedno uruchomienie potrzebuje. Jeśli powtórzymy tę operację wielokrotnie dla wielu wartości n i przeprowadzimy statystyczną analizę wyników, uzyskamy wartość średnią. Algorytm Quicksort potrzebuje około $1{,}4\ n \times \lg(n)$ porównań do posortowania *n* elementów.

Nie jest to wcale zły sposób na uzyskanie wglądu w działanie programu. Dzięki 13 wierszom kodu i kilku eksperymentom można sporo odkryć. Znane powiedzenie przypisywane pisarzom takim jak Blaise Pascal i T. S. Eliot brzmi: „Gdybym miał więcej czasu, napisałbym Ci krótszy list". My mamy czas, więc poeksperymentujemy trochę z kodem, aby napisać krótszy (i lepszy) program.

Zagramy w przyspieszanie eksperymentu, próbując zwiększyć statystyczną dokładność i wgląd w działanie programu. Jako że wewnętrzna pętla wykonuje dokładnie `u-l` porównań, możemy nieco przyspieszyć działanie programu, zliczając te porównania za pomocą pojedynczej operacji poza pętlą. Po tej zmianie algorytm Quicksort wygląda jak na listingu 3.3.

LISTING 3.3. Algorytm Quicksort po przeniesieniu inkrementacji na zewnątrz pętli

```
comps += u-l;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

Program ten sortuje tablicę i jednocześnie sprawdza liczbę potrzebnych porównań. Jeśli jednak naszym celem jest tylko zliczenie porównań, nie musimy sortować tablicy. Na listingu 3.4 zostało usunięte prawdziwe sortowanie i pozostał tylko szkielet różnych wywołań wykonywanych przez program.

LISTING 3.4. Szkielet algorytmu Quicksort zredukowany do zliczania

```
void quickcount(int l, int u)
{   int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-l;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

Program ten działa dzięki losowemu wybieraniu przez Quicksort elementu dzielącego i dzięki założeniu, że wszystkie elementy są unikatowe. Jest on wykonywany w czasie proporcjonalnym do *n*. Podczas gdy program z listingu 3.3 wymagał proporcjonalnej do *n* ilości miejsca, teraz została ona zredukowana do stosu rekurencji, który średnio jest proporcjonalny do $\lg(n)$.

Mimo że indeksy (`l` i *u*) tablicy są niezbędne w prawdziwym programie, w tej wersji szkieletu nie mają znaczenia. Można je zastąpić jedną liczbą całkowitą (`n`), która będzie określała rozmiar podtablicy do posortowania (listing 3.5).

LISTING 3.5. Szkielet algorytmu Quicksort z jednym argumentem określającym rozmiar

```
void qc(int n)
{   int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

Bardziej naturalne teraz będzie przetworzenie tej procedury do postaci funkcji zliczającej porównania (ang. *comparison count* — `cc`), która zwraca liczbę porównań użytych przez jedno wykonanie algorytmu Quicksort. Funkcję tę przedstawia listing 3.6.

LISTING 3.6. Szkielet algorytmu Quicksort zaimplementowany jako funkcja

```
int cc(int n)
{   int m;
    if (n <= 1) return 0;
    m = randint(1, n);
   return n-1 + cc(m-1) + cc(n-m);
}
```

Przykłady zamieszczone na listingach 3.4, 3.5 i 3.6 rozwiązują ten sam podstawowy problem i potrzebują na to tyle samo czasu i pamięci. Każda kolejna wersja ma poprawioną formę, dzięki czemu jest nieco bardziej przejrzysta i zwięzła od poprzedniej.

Definiując **paradoks wynalazcy** (ang. *inventor's paradox*), George Pólya oznajmia, że: „Bardziej ambitny plan może mieć więcej szans na powodzenie"[3]. Spróbujemy teraz wykorzystać ten paradoks w analizie Quicksort. Do tej pory zadawaliśmy sobie pytanie, ile porównań potrzebuje algorytm Quicksort do posortowania tablicy zawierającej *n* elementów. Teraz zadamy bardziej ambitne pytanie: ile średnio porównań potrzebuje algorytm Quicksort do posortowania losowej tablicy o rozmiarze *n*? Możemy rozszerzyć kod z listingu 3.6, aby uzyskać pseudokod widoczny na listingu 3.7.

LISTING 3.7. Średnia liczba porównań algorytmu Quicksort jako pseudokod

```
float c(int n)
    if (n <= 1) return 0
    sum = 0
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m)
    return sum/n
```

Jeśli dane wejściowe zawierają maksymalnie jeden element, Quicksort nie wykonuje żadnych porównań, jak w przykładzie z listingu 3.6. W przypadku *n* o większej wartości kod ten bierze pod uwagę każdą wartość dzielącą (od pierwszego do ostatniego elementu — każdy jest równie prawdopodobny)

[3] George Pólya, *How to solve it*, Princeton University Press, 1945 — *przyp. red.*

i określa koszt podziału w każdym z tych miejsc. Następnie kod oblicza sumę tych wartości (w ten sposób rekursywnie rozwiązując jeden problem rozmiaru `m-1` i jeden problem rozmiaru *n-m*) i dzieli ją przez *n*, uzyskując średnią.

Gdybyśmy mogli obliczyć tę liczbę, nasze eksperymenty byłyby znacznie bardziej potężne. Zamiast przeprowadzać wiele eksperymentów z jedną wartością *n* w celu oszacowania średniej, jeden eksperyment wystarczyłby do uzyskania prawdziwej średniej. Niestety, ta potęga ma swoją cenę: program działa w czasie proporcjonalnym do 3^n (interesującym ćwiczeniem jest analiza tego czasu przy użyciu technik opisanych w tym rozdziale).

Kod z listingu 3.7 potrzebuje właśnie tyle czasu, ponieważ oblicza pododpowiedzi wielokrotnie. W takim przypadku można zastosować **programowanie dynamiczne** w celu zapisywania tych pododpowiedzi, co pozwoli na uniknięcie ich ponownego obliczania. W tym przypadku wprowadzimy tablicę `t[N+1]`, w której element `t[n]` przechowuje `c(n)`, i obliczymy jej wartości w kolejności rosnącej. *N* będzie oznaczać maksymalną wartość *n*, czyli rozmiar tablicy do posortowania. Rezultat jest widoczny na listingu 3.8.

LISTING 3.8. Obliczenia algorytmu Quicksort przy użyciu programowania dynamicznego

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

Program ten jest z grubsza transkrypcją kodu z listingu 3.7, w której zastąpiono `c(n)` zapisem `t[n]`. Jego czas wykonywania jest proporcjonalny do N^2, a ilość zajmowanego miejsca do *N*. Jedną z jego zalet jest to, że po zakończeniu wykonywania tablica `t` zawiera rzeczywiste wartości średnie (a nie przybliżoną wartość przykładowych średnich) dla elementów tablicy od `0` do *N*. Dzięki analizie tych liczb można uzyskać informacje na temat funkcjonalnej formy spodziewanej liczby porównań wykonanych przez algorytm Quicksort.

Teraz uprościmy nasz program jeszcze bardziej. Najpierw przeniesiemy człon `n-1` poza pętlę, jak widać na listingu 3.9.

LISTING 3.9. Obliczenia Quicksort z kodem przeniesionym na zewnątrz pętli

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
    t[n] = n-1 + sum/n
```

Dalsze dostrajanie kodu będzie polegało na użyciu symetrii. Jeśli na przykład `n` wynosi `4`, wewnętrzna pętla oblicza następującą sumę:

```
t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]
```

W tym szeregu par pierwsze elementy zwiększają się, podczas gdy mniejsze zmniejszają. Możemy zatem sumę tę zapisać tak:

```
2 * (t[0] + t[1] + t[2] + t[3])
```

Za pomocą tej symetrii otrzymamy algorytm widoczny na listingu 3.10.

LISTING 3.10. Obliczenia Quicksort przy użyciu symetrii

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
    t[n] = n-1 + sum/n
```

Kod ten jednak również nie jest w pełni efektywny, ponieważ wielokrotnie oblicza tę samą sumę. Zamiast dodawać wszystkie poprzednie człony, możemy zmienną `sum` zainicjalizować poza pętlą i dodać następny człon. Rezultat jest widoczny na listingu 3.11.

LISTING 3.11. Obliczenia Quicksort z usuniętą wewnętrzną pętlą

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n
```

Ten niewielki program jest naprawdę użyteczny. W czasie proporcjonalnym do *N* tworzy tabelę rzeczywistych spodziewanych czasów wykonania algorytmu Quicksort dla każdej liczby całkowitej od 1 do *N*.

Kod z listingu 3.11 jest łatwy do użycia w arkuszu kalkulacyjnym, w którym wartości są natychmiast dostępne do dalszej analizy. Tabela 3.1 przedstawia początkowe wiersze.

TABELA 3.1. Wynik implementacji kodu z listingu 3.11 w arkuszu kalkulacyjnym

N	Suma	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

Pierwszy wiersz liczb w tej tabeli jest inicjalizowany za pomocą trzech stałych z kodu. W notacji arkuszy kalkulacyjnych kolejny wiersz liczb (trzeci wiersz arkusza) jest obliczany przy użyciu następujących zależności:

```
A3 = A2+1          B3 = B2 + 2*C2          C3 = A3-1 + B3/A3
```

Kopiując poprzez przeciągnięcie te (względne) odwołania w dół, można uzupełnić arkusz. Ten arkusz jest moim poważnym kandydatem na „najpiękniejszy kod, jaki kiedykolwiek napisałem” w kategorii osiągania jak najwięcej za pomocą tylko kilku wierszy kodu.

Co jednak, jeśli nie potrzebujemy tych wszystkich wartości? Gdybyśmy na przykład woleli przeanalizować tylko kilka z wartości (na przykład wszystkie potęgi cyfry 2 od 2^0 do 2^{32})? Mimo że kod z listingu 3.11 tworzy pełną tablicę `t`, używa on tylko jej najnowszej wartości.

Możemy zatem zastąpić liniową przestrzeń tablicy `t[]` stałą przestrzenią zmiennej `t`, jak na listingu 3.12.

Listing 3.12. Obliczenia Quicksort — ostateczna wersja

```
sum = 0; t = 0
for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n
```

Można następnie wstawić dodatkowy wiersz kodu w celu sprawdzenia trafności `n` i w razie potrzeby wydrukować wyniki.

Ten niewielki program jest ostatnim etapem naszej podróży. Dobrą konkluzją w odniesieniu do niej mogą być słowa Alana Perlisa: „Prostota nie występuje przed złożonością, ale jest jej następstwem”[4].

Perspektywa

Tabela 3.2 zawiera zestawienie programów analizujących Quicksort, prezentowanych w tym rozdziale.

TABELA 3.2. Ewolucja programu analizującego pracę algorytmu Quicksort

Numer przykładu	Liczba wierszy	Typ odpowiedzi	Liczba odpowiedzi	Czas trwania	Przestrzeń
2	13	Przykładowa	1	$n \times \lg(n)$	N
3	13	"	"	"	"
4	8	"	"	n	$\lg(n)$
5	8	"	"	"	"
6	6	"	"	"	"
7	6	Dokładna	"	3^N	N
8	6	"	N	N^2	N
9	6	"	"	"	"
10	6	"	"	"	"
11	4	"	"	N	"
12	4	Dokładna	N	N	1

[4] Alan Perlis, *Epigrams on Programming*, „Sigplan Notices”, Vol. 17, Issue 9 — *przyp. red.*

Każdy etap ewolucji naszego kodu był bardzo prosty. Przejście od przykładu zamieszczonego na listingu 3.6 do dokładnej odpowiedzi na listingu 3.7 jest prawdopodobnie najbardziej subtelne. Kod w miarę kurczenia się stawał się coraz szybszy. W połowie XIX wieku Robert Browning zauważył, że „mniej oznacza więcej". Ta tabela umożliwia ilościowe określenie jednego z przykładów tamtej minimalistycznej filozofii.

Widzieliśmy trzy zasadniczo różniące się typy programów. Przykłady z listingów 3.2 i 3.3 są działającymi algorytmami Quicksort przystosowanymi do zliczania porównań w trakcie sortowania prawdziwej tablicy. Listingi 3.4 do 3.6 implementują prosty model Quicksort — imitują jedno uruchomienie algorytmu, w rzeczywistości nie wykonując żadnego sortowania. Listingi 3.7 do 3.12 implementują bardziej wyrafinowany model — obliczają rzeczywistą średnią liczbę porównań, nie badania jakiegoś konkretnego uruchomienia algorytmu.

Oto podsumowanie technik zastosowanych do uzyskania każdego z programów:

- Listingi 3.2, 3.4, 3.7 — fundamentalna zmiana definicji problemu.
- Listingi 3.5, 3.6, 3.12 — nieduża zmiana definicji funkcji.
- Listing 3.8 — nowa struktura danych implementująca programowanie dynamiczne.

Techniki te są typowe. Program często można uprościć poprzez odpowiedzenie sobie na pytanie, jaki problem tak naprawdę trzeba rozwiązać oraz czy jest funkcja lepiej nadająca się do rozwiązania tego problemu.

Kiedy po raz pierwszy przedstawiłem tę analizę studentom, program w końcu skurczył się do 0 wierszy kodu i zniknął w tumanie matematycznego kurzu. Kod z listingu 3.7 można przedstawić za pomocą następującej zależności rekurencyjnej:

$$C_0 = 0 \qquad C_n = (n-1) + (1/n)\sum_{1 \le i \le n} C_{i-1} + C_{n-i}$$

Jest to dokładnie metoda zastosowana przez Hoare'a i później przedstawiona przez D. E. Knutha w jego klasycznej monografii *Sztuka programowania. Tom 3: Sortowanie i wyszukiwanie*[5]. Sztuczki programistyczne polegające na wprowadzaniu odpowiedników i zastosowaniu symetrii, dzięki którym powstał kod zaprezentowany na listingu 3.10, umożliwiły uproszczenie części rekursywnej do następującej postaci:

$$C_n = n - 1 + (2/n)\sum_{0 \le i \le n-1} C_i$$

Technika Knutha polegająca na usunięciu symbolu sumy daje w wyniku (mniej więcej) kod widoczny na listingu 3.11, który można zastąpić układem dwóch zależności rekurencyjnych z dwiema niewiadomymi.

$$C_0 = 0 \qquad S_0 = 0 \qquad S_n = S_{n-1} + 2C_{n-1} \qquad C_n = n - 1 + S_n / n$$

[5] Wydawnictwa Naukowo-Techniczne, Warszawa 2002 — *przyp. red.*

Knuth uzyskuje wynik dzięki zastosowaniu matematycznej techniki czynnika sumującego (ang. *summing factor*):

$$C_n = (n+1)(2H_{n+1} - 2) - 2n \quad \sim \quad 1{,}386 n \ln n$$

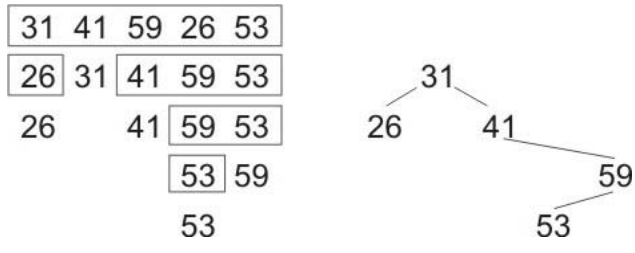
gdzie H_n oznacza n-tą liczbę harmoniczną — 1 + ½ + ⅓+ … $^1/_n$. W ten sposób gładko przeszliśmy od eksperymentowania na programie poprzez wzbogacanie go dogłębną analizą do kompletnie matematycznej analizy jego działania.

Na tej formule kończymy naszą przygodę. Poszliśmy za słynną radą Einsteina, która brzmi: „Upraszczaj wszystko jak to tylko możliwe i ani trochę bardziej".

Dodatkowa analiza

Słynne stwierdzenie Goethego mówi, że architektura to zamrożona muzyka. W dokładnie tym samym sensie twierdzę, że struktury danych to zamrożone algorytmy. Jeśli zamrozimy algorytm Quicksort, otrzymamy strukturę danych binarnego drzewa poszukiwań. Struktura ta jest zaprezentowana w publikacji Knutha. Czas jej wykonania został przeanalizowany za pomocą relacji rekurencyjnej podobnej do tej występującej w Quicksort.

Gdybyśmy chcieli przeanalizować średni koszt wstawienia elementu do binarnego drzewa wyszukiwania, moglibyśmy zacząć od kodu, który następnie wzbogacilibyśmy o zliczanie porównań. Potem moglibyśmy przeprowadzić eksperymenty na zgromadzonych danych. Następnie moglibyśmy uprościć kod (i zwiększyć jego funkcjonalność) w sposób bardzo podobny do tego z poprzedniego podrozdziału. Prostsza metoda polega na zdefiniowaniu nowego algorytmu Quicksort z użyciem metody **idealnego podziału** pozostawiającej elementy w tej samej względnej kolejności po obu stronach. Taki algorytm Quicksort jest izomorficzny z binarnymi drzewami poszukiwań, co widać na rysunku 3.1.



Rysunek 3.1. Algorytm Quicksort z idealnym podziałem i odpowiadające mu drzewo binarne poszukiwań

Ramki po lewej stronie prezentują algorytm Quicksort z idealnym podziałem w trakcie działania. Graf po prawej stronie przedstawia odpowiadające mu drzewo binarne, które zostało zbudowane z tych samych danych wejściowych. Oba te procesy wykonują nie tylko tę samą *liczbę* porównań, ale dokładnie takie same ich *zestawy*. A zatem nasza poprzednia analiza w celu sprawdzenia średniej efektywności randomizującego algorytmu Quicksort, działającego na zestawie unikatowych elementów, daje nam średnią liczbę porównań do wstawienia do binarnego drzewa wyszukiwań losowo ustawionych unikatowych elementów.

Co to jest pisanie

Tworząc kody z listingów od 3.2 do 3.12, najpierw zapisałem je w swoich notatkach, następnie na tablicy dla studentów i w końcu na kartkach tego rozdziału. Programy te powstawały stopniowo. Spędziłem dużą ilość czasu nad ich analizą i jestem przekonany o tym, że nie zawierają błędów. Jednak poza implementacją w arkuszu kalkulacyjnym przykładu z listingu 3.11 nigdy nie uruchomiłem żadnego z tych programów jako programu komputerowego.

W ciągu dwudziestu lat pracy w Bell Labs miałem okazję uczyć się od wielu nauczycieli (zwłaszcza od Briana Kernighana, którego rozdział o nauczaniu programowania pojawia się jako pierwszy w tej książce). Nauczono mnie, że pisanie programu do użytku publicznego to coś więcej niż tylko wpisywanie symboli. Po napisaniu kodu programu uruchamia się go w kilku przypadkach testowych, następnie buduje szczegółowe rusztowanie, sterowniki i bibliotekę przypadków systematycznie na nim uruchamianych. W idealnej sytuacji skompilowany kod źródłowy jest „włączany w tekst" bez interwencji człowieka. Przykład z listingu 3.1 (i wszystkie kody w książce *Perełki programowania*) napisałem właśnie w tym duchu.

Punktem honoru dla mnie było trzymanie się tytułu i nieimplementowanie przykładów z listingów 3.2 do 3.12. Prawie czterdzieści lat programowania komputerów nauczyło mnie głębokiego szacunku dla trudności tego rzemiosła (mówiąc dokładniej, panicznego strachu przed błędami). Skapitulowałem, implementując kod z listingu 3.11 w arkuszu kalkulacyjnym, i dorzuciłem dodatkową kolumnę, która dała zamkniętą formę rozwiązania. Wyobraź sobie moją radość, kiedy zobaczyłem, że dokładnie do siebie pasują! Tak więc prezentuję światu te piękne nienapisane programy z pewną dozą pewności, że są poprawne, ale w głębi będąc boleśnie świadom, że mogą zawierać jakieś nieodkryte błędy. Mam nadzieję, że głębokie piękno, które w nich widzę, nie zostanie przekreślone przez jakieś powierzchowne skazy.

Prezentując niepewnie te nienapisane programy, pocieszam się spostrzeżeniem Alana Perlisa, który powiedział: „Czy jest możliwe, że oprogramowanie nie jest podobne do niczego innego, że jest skazane na wyrzucenie, że cała filozofia polega na tym, aby postrzegać je jako mydlaną bańkę?".

Zakończenie

Piękno ma wiele źródeł. Ten rozdział koncentruje się na pięknie zdobywanym dzięki prostocie, elegancji i zwięzłości. Poniższe stwierdzenia wyrażają tę najistotniejszą myśl:

- Staraj się dodawać funkcje poprzez usuwanie kodu.
- Projektant może uznać, że osiągnął perfekcję, nie wtedy, kiedy nie pozostało już nic do dodania, ale wtedy, gdy nie można już nic odjąć (Saint-Exupéry).
- W oprogramowaniu najpiękniejszego kodu, najpiękniejszych funkcji i najpiękniejszych programów czasami w ogóle nie ma.
- Piszący szybko piszą zwięźle. Pomijają niepotrzebne słowa (Strunk i White).

- Najtańsze, najszybsze i najbardziej niezawodne komponenty systemu komputerowego to te, których nie ma (Bell).
- Dąż do robienia coraz więcej za pomocą coraz mniejszej ilości kodu.
- Gdybym miał więcej czasu, napisałbym Ci krótszy list (Pascal).
- Paradoks wynalazcy: bardziej ambitny plan może mieć więcej szans na powodzenie (Pólya).
- Prostota nie występuje przed złożonością, ale jest jej następstwem (Perlis).
- Mniej oznacza więcej (Browning).
- Upraszczaj wszystko jak to tylko możliwe i ani trochę bardziej (Einstein).
- Oprogramowanie powinno być czasami postrzegane jako mydlana bańka (Perlis).
- Szukaj piękna w prostocie.

Na tym kończy się ta lekcja. Idź zatem i postępuj, jak tu napisano.

Dla tych, którzy potrzebują bardziej konkretnych wskazówek, poniżej przedstawiam listę koncepcji podzielonych na trzy kategorie.

Analiza programów

Jednym ze sposobów na zyskanie wglądu w działanie programu jest odpowiednie wyposażenie go i uruchomienie na reprezentatywnej próbce danych, jak w przykładzie z listingu 3.2. Często jednak bardziej niż całym programem zajmujemy się jednym jego fragmentem. W tym przypadku zajmowaliśmy się tylko średnią liczbą porównań wykonywanych przez Quicksort, a pomijaliśmy wiele innych aspektów. Sedgewick[6] bada takie zagadnienia, jak wymagana przez niego przestrzeń i wiele innych komponentów wykonywania różnych wariantów Quicksort. Koncentrując się na najważniejszych problemach, możemy (przez chwilę) zapomnieć o innych aspektach programu. W jednym z moich artykułów, *A Case Study In Applied Algorithm Design*[7], opisuję, jak zetknąłem się z problemem oszacowania wydajności **heurystyki paskowej** (ang. *strip heuristic*) do znalezienia przybliżonej drogi akwizytora przez `N` punktów w określonym kwadracie. Oceniłem, że kompletny program do rozwiązania tego zadania zajmie około 100 wierszy kodu. Po kilku etapach podobnych do tych opisanych powyżej uzyskałem dwunastowierszową symulację o znacznie większej dokładności (a po zakończeniu mojej małej symulacji odkryłem, że Beardwood i inni autorzy[8] wyrazili moją symulację w postaci podwójnej liczby całkowitej, a więc rozwiązali matematycznie ten problem około dwudziestu lat wcześniej).

---

[6] Robert Sedgewick, *The Analysis of Quicksort programs*, „Acta Informatica", Vol. 7 — *przyp. red.*

[7] „IEEE Computer", Vol. 17, No. 2 — *przyp. red.*

[8] J. Beardwood, J. H. Halton, J. M. Hammersley, *The Shortest Path Through Many Points*, „Proc. Cambridge Philosophical Soc.", Vol. 55 — *przyp. red.*

Małe fragmenty kodu

Uważam, że programowanie komputerów to umiejętność praktyczna, i zgadzam się z Pólyą, iż „zdolności praktyczne nabywamy poprzez naśladownictwo i praktykę". Programiści, którzy pragną pisać piękny kod, powinni zatem czytać piękne programy i naśladować zastosowane w nich techniki we własnych. Według mnie do takich ćwiczeń najlepiej nadają się niewielkie fragmenty kodu, składające się z 10 do 25 wierszy kodu. Przygotowywanie drugiego wydania książki *Perełki programowania* wymagało mnóstwa pacy, ale było też bardzo zabawne. Implementowałem każdy fragment kodu i pracowałem nad nim, aby zredukować jego rozmiar do niezbędnego minimum. Mam nadzieję, że inni będą mieli tyle samo radości z czytania tego kodu co ja z jego pisania.

Systemy oprogramowania

Opisałem niezwykle szczegółowo jedno małe zadanie. Wydaje mi się, że świetność tych zasad nie bierze się z małych fragmentów kodu, a z dużych programów i wielkich systemów komputerowych. Parnas opisuje techniki redukcji systemu do niezbędnego minimum[9]. Stosując je, nie zapomnij rady Toma Duffa: „Podkradaj kod, kiedy tylko jest taka możliwość".

Podziękowania

Dziękuję za wnikliwe komentarze Danowi Bentleyowi, Brianowi Kernighanowi, Andy'emu Oramowi i Davidowi Weissowi.

[9] David L. Parnas, *Designing software for ease of extension and contraction*, „IEEE T. Software Engineering", Vol. 5, No. 2 — *przyp. red.*