

Wydanie II

# Wysoce wydajny C++

Opanuj sztukę  
optymalizowania działania kodu

**Björn Andrist**  
**Viktor Sehr**



Helion 

Packt>

Tytuł oryginału: C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-9708-8

Copyright © Packt Publishing 2020. First published in the English language under the title 'C++ High Performance - 2nd Edition' – (9781839216541).

Polish edition copyright © 2023 by Helion S.A.

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

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w 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 wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani a związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor raz ydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody ynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/wywy2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/wywy2.zip>

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

# Spis treści

|   |           |
|---|-----------|
| <b>Przedmowa</b>                                      | <b>11</b> |
| <b>O autorach</b>                                     | <b>13</b> |
| <b>O korektorach merytorycznych</b>                   | <b>14</b> |
| <b>Wprowadzenie</b>                                   | <b>15</b> |
| Dla kogo przeznaczona jest ta książka?                | 16        |
| Zawartość książki                                     | 16        |
| Jak najlepiej skorzystać z tej książki?               | 17        |
| Pobieranie plików z przykładowym kodem                | 18        |
| Stosowane konwencje typograficzne                     | 19        |
| <b>Rozdział 1. Krótkie wprowadzenie do języka C++</b> | <b>21</b> |
| <b>Dlaczego C++?</b>                                  | <b>21</b> |
| Bezkosztowe abstrakcje                                | 22        |
| Przenośność   | 24        |
| Odporność   | 24        |
| Język C++ dzisiaj                                     | 25        |
| <b>Porównanie C++ z innymi językami</b>               | <b>25</b> |
| Konkurencyjne języki i wydajność                      | 25        |
| Mechanizmy języka C++ niezwiązane z wydajnością       | 27        |
| Wady języka C++                                       | 33        |
| <b>Biblioteki i kompilatory używane w tej książce</b> | <b>33</b> |
| <b>Podsumowanie</b>                                   | <b>34</b> |

|  |            |
|--|------------|
| <b>Rozdział 2. Podstawowe techniki języka C++</b>                    | <b>35</b>  |
| <b>Automatyczne wykrywanie typów za pomocą słowa kluczowego auto</b> | <b>36</b>  |
| Stosowanie słowa kluczowego auto w sygnaturach funkcji               | 36         |
| Stosowanie słowa kluczowego auto dla zmiennych                       | 38         |
| <b>Semantyka przenoszenia</b>  | <b>41</b>  |
| Tworzenie przez kopiowanie, wymienianie i przenoszenie               | 42         |
| Pozyskiwanie zasobów i reguła pięciu                                 | 44         |
| Zmienne nazwane i r-wartości   | 47         |
| Domyślna semantyka przenoszenia i reguła zera                        | 48         |
| Stosowanie modyfikatora && do funkcji składowych klas                | 52         |
| Nie przenos obiektów, jeśli kopiowanie i tak jest pomijane           | 52         |
| Jeśli to możliwe, stosuj przekazywanie przez wartość                 | 53         |
| <b>Projektowanie interfejsów z obsługą błędów</b>                    | <b>55</b>  |
| Kontrakty  | 56         |
| Obsługa błędów   | 60         |
| <b>Obiekty funkcyjne i wyrażenia lambda</b>                          | <b>66</b>  |
| Podstawowa składnia lambda w języku C++                              | 66         |
| Klauzula przechwytywania   | 67         |
| Przypisywanie do lambda wskaźników do funkcji języka C               | 73         |
| Typy lambda  | 73         |
| Lambdy i typ std::function   | 73         |
| Generyczne lambdy  | 77         |
| <b>Podsumowanie</b>  | <b>78</b>  |
| <b>Rozdział 3. Analizowanie i pomiar wydajności</b>                  | <b>79</b>  |
| <b>Złożoność asymptotyczna i notacja dużego O</b>                    | <b>80</b>  |
| Tempo wzrostu  | 84         |
| Zamortyzowana złożoność czasowa                                      | 85         |
| <b>Co mierzyć i w jaki sposób?</b>                                   | <b>88</b>  |
| Aspekty wydajności   | 90         |
| Przyspieszanie wykonywania kodu                                      | 91         |
| Liczniki wydajności  | 91         |
| Testy wydajności — dobre praktyki                                    | 92         |
| <b>Poznaj kod i znajdź hot spoty</b>                                 | <b>93</b>  |
| Profilery z instrumentacją   | 94         |
| Profilery z próbkowaniem   | 96         |
| <b>Mikrotesty</b>  | <b>98</b>  |
| Prawo Amdahla  | 99         |
| Pułapki związane z mikrotestami                                      | 100        |
| Przykład ilustrujący mikrotesty                                      | 101        |
| <b>Podsumowanie</b>  | <b>106</b> |
| <b>Rozdział 4. Struktury danych</b>                                  | <b>107</b> |
| <b>Cechy pamięci w komputerach</b>                                   | <b>107</b> |
| <b>Kontenery z biblioteki standardowej</b>                           | <b>111</b> |
| Kontenery sekwencyjne  | 112        |
| Kontenery asocjacyjne  | 117        |
| Adaptory kontenerów  | 121        |

|   |            |
|---|------------|
| <b>Używanie widoków</b>   | <b>123</b> |
| Unikanie kopiowania dzięki typowi <code>string_view</code>                                  | 124        |
| Unikanie utraty informacji o długości tablic dzięki typowi <code>std::span</code>           | 125        |
| <b>Uwagi na temat wydajności</b>  | <b>126</b> |
| Zapewnianie równowagi między gwarancjami złożoności<br>a dodatkowymi kosztami               | 126        |
| Znajomość i stosowanie odpowiednich funkcji API   | 127        |
| <b>Tablice równoległe</b>   | <b>129</b> |
| <b>Podsumowanie</b>   | <b>135</b> |
| <b>Rozdział 5. Algorytmy</b>  | <b>136</b> |
| <b>Wprowadzenie do algorytmów z biblioteki standardowej</b>                                 | <b>137</b> |
| Ewolucja algorytmów z biblioteki standardowej   | 137        |
| Rozwiązywanie codziennych problemów   | 138        |
| <b>Iteratory i zakresy</b>  | <b>144</b> |
| Wprowadzenie do iteratorów  | 144        |
| Wartość wartownika i iteratory zakończone   | 145        |
| Zakresy   | 146        |
| Kategorie iteratorów  | 147        |
| <b>Cechy algorytmów standardowych</b>   | <b>149</b> |
| Algorytmy nie zmieniają wielkości kontenera   | 149        |
| Algorytmy zwracające dane wyjściowe wymagają zaalokowanych danych                           | 150        |
| Algorytmy domyślnie używają funkcji <code>operator==()</code> i <code>operator&lt;()</code> | 152        |
| W algorytmach z ograniczeniami używane są projekcje   | 153        |
| Algorytmy wymagają, aby operatory przenoszące nie zgłaszały wyjątków                        | 153        |
| Algorytmy mają gwarantowaną złożoność   | 154        |
| Algorytmy działają równie dobrze jak analogiczne funkcje z bibliotek języka C               | 155        |
| <b>Pisanie i stosowanie algorytmów generycznych</b>   | <b>155</b> |
| Algorytmy niegeneryczne   | 156        |
| Algorytmy generyczne  | 156        |
| Struktury danych, które mogą być używane przez algorytmy generyczne                         | 157        |
| <b>Dobre praktyki</b>   | <b>159</b> |
| Stosowanie algorytmów z ograniczeniami  | 159        |
| Sortowanie tylko tych danych, które będą pobierane  | 159        |
| Stosowanie algorytmów standardowych zamiast surowych pętli <code>for</code>                 | 162        |
| Unikanie tworzenia kopii kontenera  | 168        |
| <b>Podsumowanie</b>   | <b>169</b> |
| <b>Rozdział 6. Zakresy i widoki</b>   | <b>170</b> |
| <b>Powody powstania biblioteki <code>Ranges</code></b>                                      | <b>170</b> |
| Ograniczenia biblioteki <code>Algorithm</code>  | 171        |
| <b>Widoki z biblioteki <code>Ranges</code></b>  | <b>173</b> |
| Widoki można łączyć w łańcuch   | 174        |
| Widoki zakresów i adaptory zakresów   | 175        |
| Widoki są zakresami bez własności elementów<br>oferującymi gwarancje złożoności             | 176        |
| Widoki nie modyfikują podstawowego kontenera  | 176        |
| Widoki można materializować do postaci kontenerów   | 177        |
| Widoki są przetwarzane leniwie  | 178        |

|   |            |
|---|------------|
| <b>Widoki z biblioteki standardowej</b>                                 | <b>179</b> |
| Widoki dla zakresów   | 179        |
| Jeszcze o typach <code>std::string_view</code> i <code>std::span</code> | 182        |
| <b>Przyszłość biblioteki Ranges</b>                                     | <b>183</b> |
| <b>Podsumowanie</b>   | <b>183</b> |
| <b>Rozdział 7. Zarządzanie pamięcią</b>                                 | <b>184</b> |
| <b>Pamięć w komputerze</b>  | <b>185</b> |
| Wirtualna przestrzeń adresowa   | 185        |
| Strony pamięci  | 185        |
| Migotanie   | 186        |
| <b>Pamięć procesu</b>   | <b>187</b> |
| Pamięć na stosie  | 188        |
| Pamięć na stercie   | 191        |
| <b>Obiekty w pamięci</b>  | <b>192</b> |
| Tworzenie i usuwanie obiektów   | 192        |
| Wyrównanie pamięci  | 195        |
| Dopełnienie   | 199        |
| <b>Własność pamięci</b>   | <b>201</b> |
| Pośrednie zarządzanie zasobami  | 202        |
| Kontenery   | 204        |
| Inteligentne wskaźniki  | 204        |
| <b>Optymalizacja małych obiektów</b>                                    | <b>207</b> |
| <b>Niestandardowe zarządzanie pamięcią</b>                              | <b>210</b> |
| Tworzenie puli pamięci  | 211        |
| Niestandardowy alokator pamięci   | 214        |
| Stosowanie polimorficznych alokatorów pamięci                           | 218        |
| Implementowanie niestandardowego zasobu pamięciowego                    | 222        |
| <b>Podsumowanie</b>   | <b>223</b> |
| <b>Rozdział 8. Programowanie z przetwarzaniem w czasie kompilacji</b>   | <b>224</b> |
| <b>Wprowadzenie do metaprogramowania z użyciem szablonów</b>            | <b>225</b> |
| Tworzenie szablonów   | 226        |
| Stosowanie liczb całkowitych jako parametrów szablonu                   | 228        |
| Tworzenie specjalizacji szablonu  | 228        |
| Jak kompilator przetwarza funkcję szablonową?                           | 229        |
| Skrócone szablony funkcji   | 229        |
| Pobieranie typu zmiennej za pomocą specyfikatora <code>decltype</code>  | 230        |
| <b>Cechy typów</b>  | <b>231</b> |
| Kategorie cech typów  | 231        |
| Stosowanie cech typów   | 232        |
| <b>Programowanie z użyciem stałych wyrażeń</b>                          | <b>233</b> |
| Działanie funkcji ze słowem kluczowym <code>constexpr</code>            |            |
| w czasie wykonywania programu   | 234        |
| Deklarowanie funkcji natychmiastowych                                   |            |
| za pomocą słowa kluczowego <code>constexpr</code>                       | 235        |
| Instrukcja <code>if constexpr</code>                                    | 235        |
| Sprawdzanie błędów programistycznych w czasie kompilacji                | 239        |

|  |            |
|--|------------|
| <b>Ograniczenia i koncepty</b>   | <b>240</b> |
| Szablon Point2D bez ograniczeń   | 241        |
| Omówienie składni ograniczeń i konceptów   | 243        |
| Wersja szablonu Point2D z ograniczeniami   | 246        |
| Dodawanie ograniczeń do kodu   | 247        |
| Koncepty w bibliotece standardowej   | 248        |
| <b>Praktyczne przykłady metaprogramowania</b>                                    | <b>248</b> |
| Przykład nr 1: tworzenie generycznej bezpiecznej funkcji rzutowania              | 248        |
| Przykład nr 2: obliczanie skrótów łańcuchów znaków w czasie kompilacji           | 251        |
| <b>Podsumowanie</b>  | <b>256</b> |
| <b>Rozdział 9. Podstawowe narzędzia</b>  | <b>257</b> |
| <b>Reprezentowanie wartości opcjonalnych za pomocą typu std::optional</b>        | <b>258</b> |
| Opcjonalne zwracane wartości   | 258        |
| Opcjonalne zmienne składowe  | 259        |
| Unikanie pustych stanów w wyliczeniach   | 259        |
| Sortowanie i porównywanie wartości typu std::optional                            | 260        |
| <b>Kolekcje niejednorodne o stałej wielkości</b>                                 | <b>261</b> |
| Stosowanie typu std::pair  | 261        |
| Typ std::tuple   | 262        |
| <b>Kolekcje niejednorodne o dynamicznie zmienianej wielkości</b>                 | <b>270</b> |
| Typ std::variant   | 271        |
| Kolekcje niejednorodne z obiektami typu std::variant                             | 275        |
| Dostęp do wartości z kontenera elementów typu VariantType                        | 276        |
| <b>Praktyczne przykłady</b>  | <b>277</b> |
| Przykład nr 1: projekcje i operatory porównania                                  | 277        |
| Przykład nr 2: refleksja   | 278        |
| <b>Podsumowanie</b>  | <b>281</b> |
| <b>Rozdział 10. Obiekty pośredniczące i leniwe przetwarzanie</b>                 | <b>282</b> |
| <b>Wprowadzenie do leniwego przetwarzania i obiektów pośredniczących</b>         | <b>282</b> |
| Przetwarzanie leniwe i przetwarzanie zachłanne                                   | 283        |
| Obiekty pośredniczące  | 284        |
| <b>Stosowanie obiektów pośredniczących do zapobiegania tworzeniu obiektów</b>    | <b>284</b> |
| Porównywanie scalanych łańcuchów znaków z wykorzystaniem obiektu pośredniczącego | 284        |
| Implementowanie obiektu pośredniczącego  | 285        |
| Modyfikator r-wartości   | 286        |
| Przypisywanie połączonej wartości do obiektu pośredniczącego                     | 287        |
| Ocena wydajności   | 287        |
| <b>Odraczanie obliczania kwadratów</b>   | <b>288</b> |
| Prosta klasa reprezentująca wektory dwuwymiarowe                                 | 288        |
| Obliczenia matematyczne  | 289        |
| Implementowanie klasy LengthProxy  | 291        |
| Porównywanie długości za pomocą klasy LengthProxy                                | 292        |
| Obliczanie długości za pomocą klasy LengthProxy                                  | 293        |
| Ocena wydajności   | 294        |
| <b>Pomysłowe przeciążanie operatorów i obiekty pośredniczące</b>                 | <b>295</b> |
| Operator potoku jako metoda rozszerzająca  | 296        |
| <b>Podsumowanie</b>  | <b>297</b> |

|  |            |
|--|------------|
| <b>Rozdział 11. Współbieżność</b>  | <b>298</b> |
| <b>Podstawy współbieżności</b>   | <b>299</b> |
| <b>Co sprawia, że programowanie współbieżne jest trudne?</b>               | <b>299</b> |
| <b>Współbieżność a równoległość</b>  | <b>300</b> |
| Porcjowanie czasu  | 300        |
| Współdzielona pamięć   | 302        |
| Sytuacja wyścigu   | 303        |
| Muteks   | 305        |
| Zakleszczenie  | 306        |
| Zadania synchroniczne i zadania asynchroniczne                             | 307        |
| <b>Programowanie współbieżne w języku C++</b>                              | <b>308</b> |
| Biblioteka obsługi wątków  | 308        |
| Dodatkowe podstawowe mechanizmy synchronizacji w standardzie C++20         | 320        |
| Obsługa zmiennych atomowych w języku C++                                   | 329        |
| Model dostępu do pamięci w języku C++                                      | 337        |
| <b>Programowanie bez blokad</b>  | <b>341</b> |
| Przykład: kolejka bez blokad   | 341        |
| <b>Wskazówki dotyczące wydajności</b>                                      | <b>344</b> |
| Zapobieganie rywalizacji   | 344        |
| Unikanie operacji blokujących  | 344        |
| Liczba wątków i rdzeni procesora   | 345        |
| Priorytety wątków  | 345        |
| Koligacja wątków   | 346        |
| Niezamierzone współdzielenie   | 347        |
| <b>Podsumowanie</b>  | <b>347</b> |
| <b>Rozdział 12. Korutyny i leniwe generatory</b>                           | <b>348</b> |
| <b>Kilka przykładów uzasadniających stosowanie korutyn</b>                 | <b>349</b> |
| <b>Abstrakcja reprezentująca korutyny</b>                                  | <b>351</b> |
| Podprocedury i korutyny  | 351        |
| Wykonywanie podprocedur i korutyn w procesorze                             | 353        |
| Korutyny bezstosowe i korutyny stosowe                                     | 360        |
| Czego Czytelnik nauczył się do tego miejsca?                               | 363        |
| <b>Korutyny w języku C++</b>   | <b>363</b> |
| Co zostało uwzględnione w standardzie języka C++ (i co zostało pominięte)? | 364        |
| Co sprawia, że funkcja w języku C++ jest korutyną?                         | 365        |
| Minimalny, lecz kompletny przykład   | 366        |
| Alokowanie stanu korutyny  | 371        |
| Zapobieganie wiszącym referencjom  | 372        |
| Obsługa błędów   | 376        |
| Punkty konfiguracyjne  | 377        |
| <b>Generatory</b>  | <b>377</b> |
| Implementowanie generatora   | 377        |
| Stosowanie klasy Generator   | 381        |
| Praktyczny przykład zastosowania generatorów                               | 387        |
| <b>Wydajność</b>   | <b>393</b> |
| <b>Podsumowanie</b>  | <b>393</b> |



|   |            |
|---|------------|
| <b>Rozdział 13. Programowanie asynchroniczne z użyciem korutyn</b>  | <b>395</b> |
| <b>Jeszcze o typach awaitable</b>                                   | <b>396</b> |
| Automatyczne punkty wstrzymywania                                   | 397        |
| <b>Implementowanie prostego typu reprezentującego zadanie</b>       | <b>398</b> |
| Obsługa zwracanych wartości i wyjątków                              | 400        |
| Wznawianie oczekującej korutyny                                     | 401        |
| Dodawanie obsługi zadań zwracających void                           | 403        |
| Synchroniczne oczekiwanie na ukończenie zadania                     | 404        |
| Testowanie asynchronicznych zadań z użyciem funkcji sync_wait()     | 408        |
| <b>Tworzenie nakładki na API opartym na wywołaniach zwrotnych</b>   | <b>409</b> |
| <b>Współbieżny serwer zbudowany za pomocą biblioteki Boost.Asio</b> | <b>412</b> |
| Implementowanie serwera   | 412        |
| Uruchamianie serwera i nawiązywanie z nim połączenia                | 414        |
| Co osiągnęliśmy za pomocą serwera (i czego wciąż brakuje)?          | 415        |
| <b>Podsumowanie</b>   | <b>416</b> |
| <b>Rozdział 14. Algorytmy równoległe</b>                            | <b>417</b> |
| <b>Znaczenie równoległości</b>                                      | <b>418</b> |
| <b>Algorytmy równoległe</b>   | <b>418</b> |
| Ocena algorytmów równoległych                                       | 418        |
| Jeszcze o prawie Amdahla  | 419        |
| Implementowanie równoległego algorytmu std::transform()             | 420        |
| Implementowanie równoległej wersji algorytmu std::count_if()        | 429        |
| Implementowanie równoległej wersji algorytmu std::copy_if()         | 430        |
| <b>Algorytmy równoległe z biblioteki standardowej</b>               | <b>436</b> |
| Strategie wykonywania   | 436        |
| Obsługa wyjątków  | 440        |
| Dodatki i zmiany w algorytmach równoległych                         | 441        |
| Zrównoleglanie pętli for opartej na indeksie                        | 443        |
| <b>Wykonywanie algorytmów w procesorze graficznym</b>               | <b>444</b> |
| <b>Podsumowanie</b>   | <b>445</b> |



# Analizowanie i pomiar wydajności

Ponieważ jest to książka na temat pisania wydajnego kodu w języku C++, trzeba omówić podstawy pomiaru wydajności oprogramowania i szacowania złożoności algorytmicznej. Większość zagadnień opisywanych w tym rozdziale nie ogranicza się do języka C++ i ma zastosowanie w każdym problemie, w którym ważna jest wydajność.

Dalej wyjaśniamy, jak szacować złożoność algorytmiczną za pomocą notacji dużego O. Jest to podstawowa metoda stosowana przy wyborze algorytmów i struktur danych z biblioteki standardowej języka C++. Jeśli pierwszy raz masz styczność z notacją dużego O, opanowanie tego fragmentu może zająć Ci trochę czasu. Jednak nie poddawaj się! Jest to bardzo istotna kwestia, potrzebna do zrozumienia reszty książki i, co ważniejsze, do tego, by zostać programistą znającym się na wydajności. Jeśli szukasz bardziej formalnego lub bardziej praktycznego wprowadzenia do omawianych tu zagadnień, dostępnych jest wiele poświęconych im książek i materiałów internetowych. Z kolei jeżeli już opanowałeś notację dużego O i wiesz, czym jest koszt zamortyzowany, możesz pominąć następny podrozdział i przejść do dalszych fragmentów tego rozdziału.

W tym rozdziale omawiamy następujące tematy:

- szacowanie złożoności algorytmicznej za pomocą notacji dużego O;
- zalecany proces optymalizowania kodu, który pozwala uniknąć dopracowywania kodu, jeśli nie ma do tego dobrych powodów;
- profilery procesorów (czy one są i dlaczego należy je stosować);
- mikrotesty.

Zacniemy od przyjrzenia się szacowaniu złożoności algorytmicznej za pomocą notacji dużego O.

# Złożoność asymptotyczna i notacja dużego O

Zwykle istnieje więcej niż jeden sposób na rozwiązanie problemu, a jeśli wydajność jest istotna, należy najpierw skoncentrować się na wysokopoziomowych optymalizacjach, wybierając odpowiednie algorytmy i struktury danych. Przydatnym sposobem oceny i porównywania algorytmów jest analiza ich asymptotycznej złożoności obliczeniowej, czyli tego, jak czas wykonywania lub zużycie pamięci rosną, gdy rośnie wielkość danych wejściowych. Ponadto w bibliotece standardowej języka C++ określona jest złożoność asymptotyczna wszystkich kontenerów i algorytmów, co oznacza, że podstawowa znajomość omawianych tu tematów jest niezbędna do korzystania z tej biblioteki. Jeśli już dobrze rozumiesz złożoność algorytmów i notację dużego O, możesz swobodnie pominąć ten podrozdział.

Zacniemy od przykładu. Załóżmy, że chcesz napisać algorytm, który zwraca wartość `true`, jeśli znajdzie określony klucz w tablicy, a w przeciwnym razie zwraca wartość `false`. Aby sprawdzić, jak ten algorytm działa dla tablic o różnej wielkości, należy przeanalizować czas wykonywania go jako funkcję od wielkości danych wejściowych:

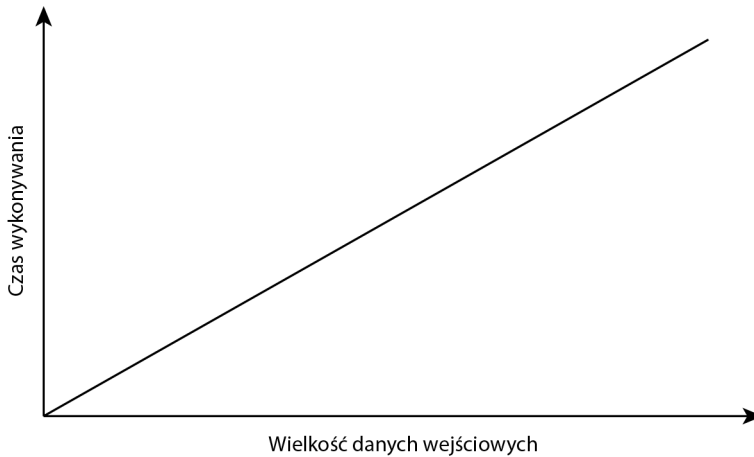
```
bool linear_search(const std::vector<int>& vals, int key) noexcept {
    for (const auto& v : vals) {
        if (v == key) {
            return true;
        }
    }
    return false;
}
```

Jest to prosty algorytm. Iteracyjnie pobiera elementy tablicy i porównuje każdy z nich z szukanym kluczem. Jeśli masz szczęście, algorytm znajdzie klucz na początku tablicy i natychmiast zwróci sterowanie. Możliwe jednak, że trzeba będzie sprawdzić w pętli całą tablicę, a klucz w ogóle nie zostanie znaleziony. Jest to przypadek pesymistyczny dla algorytmu i zwykle to właśnie taki scenariusz należy przeanalizować.

Co się jednak dzieje z czasem wykonywania, gdy wielkość danych wejściowych rośnie? Załóżmy, że podwajamy wielkość tablicy. W najgorszym przypadku trzeba sprawdzić wszystkie elementy tablicy, co podwaja czas wykonywania. Wygląda na to, że występuje liniowa zależność między wielkością danych wejściowych a czasem wykonywania. Jest to liniowe tempo wzrostu pokazane na rysunku 3.1.

Teraz przyjrzyj się następnemu algorytmowi:

```
struct Point {
    int x_{};
    int y_{};
};
```



Rysunek 3.1. Liniowe tempo wzrostu

```
bool linear_search(const std::vector<Point>& a, const Point& key) {
    for (size_t i = 0; i < a.size(); ++i) {
        if (a[i].x_ == key.x_ && a[i].y_ == key.y_) {
            return true;
        }
    }
    return false;
}
```

Tutaj porównujemy punkty zamiast liczb całkowitych i używamy operatora indeksu do dostępu do każdego elementu. Jak te zmiany wpływają na czas wykonywania algorytmu? Bezwzględny czas wykonywania będzie prawdopodobnie wyższy w porównaniu z pierwszym algorytmem, ponieważ kod wykonuje więcej pracy (porównanie punktów wymaga sprawdzenia dla każdego elementu tablicy dwóch liczb całkowitych zamiast tylko jednej). Jednak w tym miejscu interesuje nas tempo wzrostu, a jeśli narysujemy wykres czasu wykonywania od wielkości danych wejściowych, ponownie uzyskamy linię prostą, tak jak na rysunku 3.1.

W ramach ostatniego przykładu szukania liczb całkowitych sprawdźmy, czy można opracować lepszy algorytm przy założeniu, że elementy tablicy są posortowane. Pierwszy algorytm działa niezależnie od kolejności elementów, jeśli jednak wiadomo, że są one posortowane, można zastosować wyszukiwanie binarne. W tym algorytmie najpierw sprawdzany jest środkowy element, aby ustalić, czy należy kontynuować przeszukiwanie w pierwszej, czy w drugiej połowie tablicy. Dla uproszczenia indeksy `high`, `low` i `mid` są typu `int` i wymagają rzutowania za pomocą operatora `static_cast` (lepszym rozwiązaniem byłoby zastosowanie iteratorów, które omawiamy w dalszych rozdziałach). Oto kod algorytmu:

```
bool binary_search(const std::vector<int>& a, int key) {
    auto low = 0;
    auto high = static_cast<int>(a.size()) - 1;
    while (low <= high) {
        const auto mid = std::midpoint(low, high); // C++20
    }
}
```

```

    if (a[mid] < key) {
        low = mid + 1;
    } else if (a[mid] > key) {
        high = mid - 1;
    } else {
        return true;
    }
}
return false;
}

```

Widać tu, że trudniej jest napisać ten algorytm niż algorytm prostego wyszukiwania liniowego. Nowy algorytm szuka określonego klucza, *zgadując*, że znajduje się on pośrodku tablicy. Jeśli go tam nie ma, algorytm porównuje klucz ze środkowym elementem, aby ustalić, w której połowie tablicy należy kontynuować poszukiwanie klucza. Tak więc w każdej iteracji wielkość tablicy do sprawdzenia jest zmniejszana o połowę.

Załóżmy, że wywołujemy funkcję `binary_search()` dla tablicy zawierającej 64 elementy. W pierwszej iteracji odrzucane są 32 elementy, w następnej 16 elementów, w kolejnej 8 elementów itd. — aż do momentu znalezienia klucza lub wyczerpania się elementów do porównywania. Dla danych wejściowych obejmujących 64 elementy potrzebnych jest maksymalnie 7 iteracji pętli. Co się stanie, jeśli *podwoimy wielkość tablicy wejściowej* do 128 elementów? Ponieważ w każdej iteracji wielkość tablicy jest zmniejszana o połowę, oznacza to, że potrzebna będzie tylko *jedna dodatkowa iteracja pętli*. Tempo wzrostu nie jest już liniowe, lecz logarytmiczne. Jeśli zmierzysz czas wykonywania funkcji `binary_search()`, zobaczysz, że tempo wzrostu wygląda mniej więcej tak jak na rysunku 3.2.



Rysunek 3.2. Logarytmiczne tempo wzrostu

Na komputerze jednego z autorów szybki pomiar trzech opisanych algorytmów wywołanych 10 000 razy dla danych wejściowych o różnej wielkości ( $n$ ) dał wyniki pokazane w tabeli 3.1.

Tabela 3.1. Porównanie różnych wersji algorytmu wyszukiwania

| Algorytm  | n = 10  | n = 1000 | n = 100 000 |
|---|---------|----------|-------------|
| Wyszukiwanie liniowe wartości typu <code>int</code>   | 0,04 ms | 4,7 ms   | 458 ms      |
| Wyszukiwanie liniowe wartości typu <code>Point</code> | 0,07 ms | 6,7 ms   | 725 ms      |
| Wyszukiwanie binarne wartości typu <code>int</code>   | 0,03 ms | 0,08 ms  | 0,16 ms     |

Gdy porównasz algorytmy 1. i 2., zobaczysz, że sprawdzanie punktów zamiast liczb całkowitych wymaga więcej czasu, ale rząd wielkości tego czasu pozostaje taki sam nawet przy rosnącej wielkości danych wejściowych. Gdy porównujesz wszystkie trzy algorytmy przy rosnącej wielkości danych wejściowych, ważne jest ich tempo wzrostu. Wykorzystując fakt, że tablica jest posortowana, można zaimplementować funkcję wyszukiwania z bardzo niewielką liczbą iteracji pętli. Dla dużych tablic wyszukiwanie binarne jest prawie „darmowe” w porównaniu z liniowym przeszukiwaniem tablicy.

Zwykle nie warto przeznaczać czasu na dostrajanie kodu przed upewnieniem się, że wybrane zostały algorytmy i struktury danych odpowiednie dla danego problemu.

Czy nie byłoby wygodnie móc wyrazić tempo wzrostu algorytmów w sposób, który pomógłby w wyborze algorytmu? Właśnie to umożliwia notacja dużego  $O$ .

Oto nieformalna definicja:

Jeśli  $f(n)$  jest funkcją określającą czas wykonywania algorytmu dla danych wejściowych o wielkości  $n$ , to  $f(n)$  ma złożoność  $O(g(n))$ , jeśli istnieje stała  $k$ , dla której  $f(n) \leq k \times g(n)$ .

To oznacza, że można powiedzieć, iż złożoność czasowa obu wersji funkcji `linear_search()` (dla liczb całkowitych i dla punktów) wynosi  $O(n)$ , a złożoność czasowa funkcji `binary_search()` to  $O(\log n)$ .

W praktyce gdy chcesz ustalić złożoność  $O$  funkcji, należy wyeliminować wszystkie wyrazy oprócz wyrazu o najwyższym tempie wzrostu, a następnie usunąć wszystkie stałe. Na przykład jeśli algorytm ma złożoność czasową  $f(n) = 4n^2 + 30n + 100$ , należy wybrać wyraz o najwyższym tempie wzrostu, czyli  $4n^2$ . Następnie trzeba usunąć stałą 4, co daje w wyniku  $n^2$ . Można więc powiedzieć, że algorytm ma złożoność  $O(n^2)$ . Ustalenie złożoności czasowej algorytmu bywa trudne, jednak im częściej będziesz się nad tym zastanawiać w trakcie pisania kodu, tym łatwiej będzie Ci ją ocenić. Przeważnie wystarczy prześledzić pętlę i rekurencyjne wywołania funkcji.

Spróbujmy teraz ustalić złożoność czasową następującego algorytmu sortowania:

```
void insertion_sort(std::vector<int>& a) {
    for (size_t i = 1; i < a.size(); ++i) {
        auto j = i;
```

```

while (j > 0 && a[j-1] > a[j]) {
    std::swap(a[j], a[j-1]);
    --j;
}
}
}

```

Wielkość danych wejściowych to wielkość tablicy. Czas wykonywania można oszacować, sprawdzając pętle, które iteracyjnie pobierają wszystkie elementy. Najpierw działa pętla zewnętrzna, która iteracyjnie pobiera  $n - 1$  elementów. Pętla wewnętrzna działa inaczej. Przy pierwszym wykonaniu pętli `while` zmienna `j` jest równa 1 i pętla wykonuje tylko jedną iterację. W następnym powtórzeniu `j` ma początkowo wartość 2 i zmniejsza się do 0. W każdej kolejnej iteracji zewnętrznej pętli `for` pętla wewnętrzna musi wykonywać coraz więcej pracy. W ostatnim powtórzeniu `j` ma początkowo wartość  $n - 1$ , co oznacza, że w najgorszym przypadku funkcja `swap()` jest wykonywana  $1 + 2 + 3 + \dots + (n - 1)$  razy. Można to zapisać w kategoriach  $n$ , zauważając, że mamy do czynienia z ciągiem arytmetycznym. Suma tego ciągu wynosi:

$$1 + 2 + \dots + k = \frac{k(k + 1)}{2}$$

Jeśli więc  $k = (n - 1)$ , złożoność czasowa przedstawionego algorytmu sortowania wynosi:

$$\frac{(n - 1)(n - 1 + 1)}{2} = \frac{n(n - 1)}{2} = \frac{n^2 - n}{2} = (1/2)n^2 - (1/2)n$$

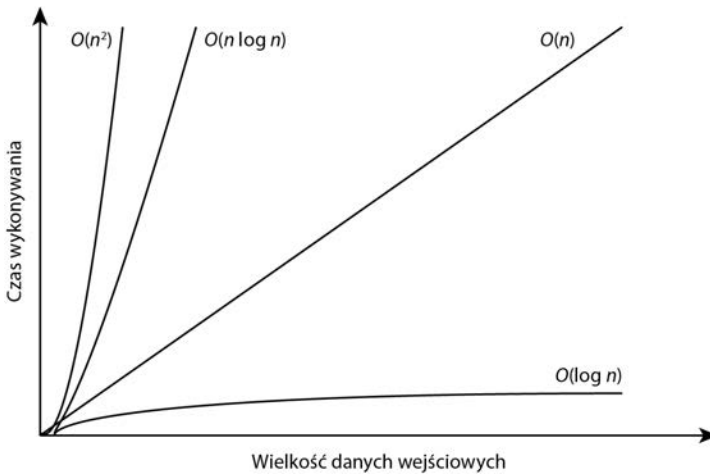
Złożoność  $O$  tej funkcji można ustalić, eliminując najpierw wszystkie wyrazy oprócz tego o najwyższym tempie wzrostu, co daje  $(1/2)n^2$ . Następnie należy usunąć stałą,  $1/2$ , i w efekcie stwierdzamy, że czas wykonywania przedstawionego algorytmu sortowania wynosi  $O(n^2)$ .

## Tempo wzrostu

Wcześniej wspomnieliśmy, że pierwszym krokiem przy szacowaniu złożoności  $O$  funkcji jest usunięcie wszystkich wyrazów oprócz tego o najwyższym tempie wzrostu. Aby było to możliwe, warto poznać tempo wzrostu niektórych często używanych funkcji. Na rysunku 3.3 przedstawiliśmy na wykresie niektóre z najczęściej spotykanych funkcji.

Tempo wzrostu jest niezależne od maszyny, stylu pisania kodu itd. Gdy dwa algorytmy mają różne tempo wzrostu, algorytm o niższym tempie wzrostu zawsze będzie wydajniejszy dla odpowiednio dużych danych wejściowych. Zobacz, co się dzieje z czasem wykonywania dla różnego tempa wzrostu (zakładamy tu, że wykonanie 1 jednostki pracy zajmuje 1 ms). W tabeli 3.2 opisane są funkcje tempa wzrostu, ich zwyczajowo stosowane nazwy i różne wielkości danych wejściowych,  $n$ .





Rysunek 3.3. Porównanie tempa wzrostu dla funkcji

Tabela 3.2. Bezwzględny czas wykonywania dla różnego tempa wzrostu i różnych wielkości danych wejściowych

| Złożoność $O$ | Nazwa                                       | $n = 10$ | $n = 50$   | $n = 1000$                |
|---------------|---|----------|------------|---------------------------|
| $O(1)$        | Stała                                       | 0,001 s  | 0,001 s    | 0,001 s                   |
| $O(\log n)$   | Logarytmiczna                               | 0,003 s  | 0,006 s    | 0,01 s                    |
| $O(n)$        | Liniowa                                     | 0,01 s   | 0,05 s     | 1 s                       |
| $O(n \log n)$ | Liniowo-<br>logarytmiczna<br>lub $n \log n$ | 0,03 s   | 0,3 s      | 10 s                      |
| $O(n^2)$      | Kwadratowa                                  | 0,1 s    | 2,5 s      | 16,7 min                  |
| $O(2^n)$      | Wykładnicza                                 | 1 s      | 35 700 lat | $3,4 \times 10^{290}$ lat |

Zauważ, że liczba w prawej dolnej komórce ma 291 cyfr! Porównaj ją z wiekiem wszechświata,  $13,7 \times 10^9$  lat, który jest wyrażony tylko 11-cyfrową liczbą.

Dalej omawiamy zamortyzowaną złożoność czasową, która jest często stosowana w bibliotece standardowej języka C++.

## Zamortyzowana złożoność czasowa

Algorytm zwykle działa inaczej dla różnych danych wejściowych. Wróćmy do algorytmu, który liniowo wyszukuje elementy w tablicy. Analizowaliśmy w nim sytuację, w której klucza w ogóle nie było w tablicy. Dla tego algorytmu jest to przypadek pesymistyczny, wymagający *najwięcej* zasobów. Przypadek optymistyczny to taki, w którym używana jest *najmniejsza* ilość zasobów. Przypadek średni określa ilość zasobów potrzebną algorytmowi średnio dla różnych danych wejściowych.

W bibliotece standardowej dla funkcji operujących na kontenerach przeważnie podawany jest *zamortyzowany czas wykonywania*. Jeśli algorytm ma stały amortyzowany czas wykonywania, oznacza to, że w prawie wszystkich sytuacjach ma złożoność  $O(1)$ . Wyjątkiem są bardzo nieliczne scenariusze, w których algorytm działa dłużej. Na pozór amortyzowany czas wykonywania można pomylić ze średnim czasem wykonywania, jednak, jak się przekonasz, nie są one identyczne.

Aby zrozumieć amortyzowaną złożoność czasową, opiszemy funkcję `std::vector::push_back()`. Załóżmy, że wektor wewnętrznie przechowuje tablicę o stałej wielkości, w której zapisane są wszystkie jego elementy. Jeśli w momencie wywołania funkcji `push_back()` w tej stałej tablicy jest miejsce na dodatkowe elementy, operacja zostanie wykonana w stałym czasie  $O(1)$ . Funkcja jest więc niezależna od tego, ile elementów już znajduje się w wektorze, jeśli tylko w wewnętrznej tablicy znajduje się miejsce na jeszcze jeden element:

```
if (internal_array.size() > size) {
    internal_array[size] = new_element;
    ++size;
}
```

Co się jednak dzieje, jeśli wewnętrzna tablica jest pełna? Jednym ze sposobów na obsługę rosnącego wektora jest utworzenie nowej pustej wewnętrznej tablicy o większej wielkości i przeniesienie wszystkich elementów z pierwotnej tablicy do nowej. To zadanie oczywiście nie jest wykonywane w stałym czasie, ponieważ konieczna jest jedna operacja przenoszenia dla każdego elementu tablicy. Złożoność wynosi więc  $O(n)$ . Jeśli uwzględnić ten przypadek pesymistyczny, oznacza to, że funkcja `push_back()` ma złożoność  $O(n)$ . Jednak jeżeli jest ona uruchamiana wielokrotnie, wiadomo, że jej kosztowne wywołania nie będą częste, dlatego stwierdzenie, że złożoność funkcji `push_back()` jest równa  $O(n)$ , byłoby pesymistyczne i nie- zbyt przydatne, gdy wiadomo, że będzie ona wykonywana wiele razy.

Zamortyzowany czas wykonywania służy do analizowania *sekwencji* operacji zamiast pojedynczych wywołań. Nadal analizujemy wtedy przypadek pesymistyczny, ale dla sekwencji operacji. Zamortyzowany czas wykonywania można obliczyć, analizując najpierw czas wykonywania całej sekwencji operacji i dzieląc go przez długość tej sekwencji. Załóżmy, że wykonujesz sekwencję  $m$  operacji o łącznym czasie działania  $T(m)$ :

$$T(m) = t_0 + t_1 + t_2 \dots + t_{m-1}$$

W tym wzorze  $t_0 = 1$ ,  $t_1 = n$ ,  $t_2 = 1$ ,  $t_3 = n$  itd. Oznacza to, że połowa operacji jest wykonywana w stałym czasie, a połowa w czasie liniowym. Łączny czas  $T$  dla wszystkich  $m$  operacji można wyrazić tak:

$$T(m) = n \cdot \frac{m}{2} + 1 \cdot \frac{m}{2} = \frac{(n+1)m}{2}$$

Zamortyzowana złożoność dla każdej operacji to łączny czas podzielony przez liczbę operacji. Wynik okazuje się równy  $O(n)$ :

$$T(m)/m = \frac{(n+1)m}{2m} = \frac{n+1}{2} = O(n)$$

Jeśli jednak można zagwarantować, że liczba kosztownych operacji różni się o rzędy wielkości od liczby operacji wykonywanych w stałym czasie, można uzyskać niższe koszty zamortyzowane. Na przykład jeżeli wiadomo, że kosztowna operacja w sekwencji  $T(n) + T(1) + T(1) + \dots$  jest wykonywana tylko raz, zamortyzowany czas wykonywania jest równy  $O(1)$ . Dlatego zamortyzowany czas wykonywania zmienia się w zależności od częstotliwości wykonywania kosztownej operacji.

Wróćmy teraz do typu `std::vector`. W standardzie języka C++ jest napisane, że funkcja `push_back()` musi mieć zamortyzowany stały czas wykonywania  $O(1)$ . Jak producenci bibliotek mogą uzyskać ten poziom? Jeśli wielkość tablicy będzie zwiększana o stałą liczbę elementów za każdym razem, gdy wektor się zapełni, sytuacja będzie podobna jak we wcześniejszym przykładzie, gdzie czas wykonywania wynosił  $O(n)$ . Nawet dla dużej stałej liczby elementów zmiana pojemności będzie wykonywana w stałych odstępach czasu. Ważnym spostrzeżeniem jest to, że wektor musi być powiększany wykładniczo, aby kosztowne operacje były wykonywane odpowiednio rzadko. Wewnątrz wektora używany jest współczynnik powiększania, a pojemność nowej tablicy jest równa pojemności pierwotnej tablicy pomnożonej przez współczynnik powiększania.

Wysoki współczynnik powiększania może prowadzić do marnowania większej ilości pamięci, ale zmniejsza częstotliwość wykonywania kosztownej operacji. Aby uprościć obliczenia, zastosujemy często używaną strategię polegającą na powiększaniu pojemności tablicy za każdym razem, gdy wektor wymaga powiększenia. Teraz można oszacować, jak częste są kosztowne wywołania. Wektor o wielkości  $n$  wymaga powiększenia wewnętrznej tablicy  $\log_2(n)$  razy, ponieważ pojemność tablicy za każdym razem jest podwajana. Przy powiększaniu tablicy trzeba przenieść wszystkie elementy, które są w niej zapisane. Przy  $i$ -tym powiększaniu tablicy konieczne jest przeniesienie  $2^i$  elementów. Dlatego jeśli wykonywanych jest  $m$  operacji `push_back()`, łączny czas wykonywania operacji powiększania wynosi:

$$T(m) = \sum_{i=1}^{\log_2(m)} 2^i$$

Jest to ciąg geometryczny, który można zapisać także w następujący sposób:

$$\frac{2 - 2^{\log_2(m)+1}}{1 - 2} = 2m - 2 = O(m)$$

Gdy podzielimy tę wartość przez długość sekwencji,  $m$ , otrzymamy zamortyzowany czas wykonywania równy  $O(1)$ .

Wspomnieliśmy wcześniej, że zamortyzowana złożoność czasowa jest często stosowana w bibliotece standardowej, dlatego warto zrozumieć przedstawione analizy. Zastanowienie się nad tym, jak można zaimplementować funkcję `push_back()`, by miała zamortyzowaną stałą złożoność,

pomogło nam zapamiętać uproszczoną wersję zamortyzowanej stałej złożoności: funkcja działa w czasie  $O(1)$  w prawie wszystkich sytuacjach z wyjątkiem bardzo nielicznych przypadków, w których ma gorszą wydajność.

To wszystko, co mamy do powiedzenia na temat złożoności asymptotycznej. Teraz zobaczysz, jak można poradzić sobie z problemami z wydajnością i skutecznie optymalizować kod.

## Co mierzyć i w jaki sposób?

Optymalizacje prawie zawsze zwiększają złożoność kodu. Wysokopoziomowe optymalizacje, na przykład odpowiedni dobór algorytmów i struktur danych, mogą skutkować bardziej jednoznacznym określeniem przeznaczenia kodu, jednak przeważnie optymalizacje zmniejszają czytelność i utrudniają konserwację kodu. Dlatego warto mieć pewność, że dodane optymalizacje rzeczywiście pozwolą uzyskać oczekiwany wzrost wydajności. Czy naprawdę konieczne jest, aby kod działał szybciej? W jakim stopniu? Czy kod rzeczywiście zużywa za dużo pamięci? Aby zrozumieć, jakie optymalizacje są możliwe, trzeba dobrze poznać wymagania dotyczące latencji, przepustowości i zużycia pamięci.

Optymalizowanie kodu jest ciekawe, ale bardzo łatwo zatracić się w tym procesie bez uzyskania żadnych mierzalnych korzyści. Ten podrozdział zaczniemy od proponowanego procesu prac, który warto stosować w trakcie dostrajania kodu:

- 1. Zdefiniuj cel.** Łatwiej jest ustalić, jak wprowadzić optymalizacje i kiedy je zakończyć, jeśli znane są ściśle zdefiniowane i ilościowe cele. W niektórych sytuacjach wymagania są znane od początku, jednak w wielu scenariuszach nie są one precyzyjnie określone. Nawet jeśli jest oczywiste, że kod działa zbyt wolno, ważne jest, aby ustalić, jaki poziom będzie wystarczająco dobry. W każdym obszarze występują jakieś limity, dlatego koniecznie ustal, jakie są związane z Twoją aplikacją. Oto kilka konkretnych przykładów:
  - czas reakcji w interaktywnych aplikacjach komunikujących się z użytkownikiem to 100 ms (zobacz stronę <https://www.nngroup.com/articles/response-times-3-important-limits/>);
  - grafika z 60 klatkami na sekundę oznacza 16 milisekund na klatkę;
  - przetwarzanie audio w czasie rzeczywistym z buforem na 128 próbek przy częstotliwości próbkowania 44,1 kHz oznacza bufor na nieco mniej niż 3 milisekundy.
- 2. Dokonaj pomiarów.** Gdy już wiadomo, co mierzyć i jakie są limity, można dokonać pomiarów i ocenić, jak aplikacja działa w danym momencie. Z kroku 1. powinno wynikać, czy istotny jest średni czas, szczytowe obciążenie, czas wczytywania itd. W niniejszym zaś kroku interesuje nas tylko pomiar ustalonego celu. W zależności od aplikacji do pomiarów można używać różnych narzędzi: od stopera po wysoce zaawansowane systemy do analizy wydajności.

3. **Znajdź wąskie gardła.** Następnie należy znaleźć wąskie gardła w aplikacji: miejsca, które są zbyt powolne i sprawiają, że aplikacja jest bezużyteczna. Na tym etapie nie ufaj swoim przeczuciom! Możliwe, że uzyskane zostały jakieś informacje dzięki pomiarowi kodu w różnych miejscach w *kroku 2*. Są one przydatne, ale zwykle trzeba przeprowadzić dodatkowe profilowanie kodu, aby znaleźć najbardziej istotne hot spoty.
4. **Postaw hipotezy.** Postaw hipotezy dotyczące możliwych sposobów na poprawę wydajności. Czy można zastosować tablicę wyszukiwania? Czy można zapisać dane w pamięci podręcznej, aby zwiększyć ogólny poziom przepustowości? Czy można zmodyfikować kod, aby kompilator mógł zastosować wektoryzację? Czy można zmniejszyć liczbę alokacji w krytycznych sekcjach kodu, ponownie wykorzystując pamięć? Wymyślanie takich pomysłów jest zazwyczaj łatwe, jeśli wiesz, że są to tylko hipotezy. Możesz się mylić — później ustalisz, czy zmiany spowodowały oczekiwaną poprawę, czy nie.
5. **Przeprowadź optymalizację.** Teraz wprowadź zmiany na podstawie hipotez zarysowanych w *kroku 4*. Nie przeznaczaj na ten krok zbyt dużo czasu. Nie twórz perfekcyjnego rozwiązania, zanim nie stwierdzisz, że przynosi ono pożądane skutki. Zachowaj gotowość do odrzucenia poszczególnych optymalizacji. Możliwe, że nie przyniosą one oczekiwanych efektów.
6. **Dokonaj oceny.** Ponownie dokonaj pomiaru. Przeprowadź dokładnie ten sam test co w *kroku 2*. i porównaj wyniki. Co udało Ci się zyskać? Jeśli zmiany nie dały żadnych korzyści, odrzuć nowy kod i wróć do *kroku 4*. Jeżeli optymalizacja przyniosła pozytywne skutki, zastanów się, czy są one wystarczające, aby uzasadnić dodatkowe prace nad nią. Jak skomplikowana jest optymalizacja? Czy jest warta zachodu? Czy poprawa wydajności jest uniwersalna, czy występuje tylko w określonych przypadkach lub systemach? Czy optymalizacja jest łatwa w utrzymaniu? Czy można ją poddać hermetyzacji, czy może pojawia się w wielu miejscach kodu bazowego? Jeśli nie możesz uzasadnić wprowadzenia danej optymalizacji, wróć do *kroku 4*. W przeciwnym razie przejdź do ostatniego kroku.
7. **Przeprowadź refaktoryzację.** Jeśli zgodnie ze wskazówkami z *kroku 5*. nie poświęciłeś zbyt dużo czasu na pisanie na samym początku perfekcyjnego kodu, pora przeprowadzić refaktoryzację optymalizacji, aby zwiększyć przejrzystość kodu. Optymalizacje prawie zawsze wymagają jakichś komentarzy wyjaśniających, dlaczego kod jest napisany w nietypowy sposób.

Stosowanie tego procesu gwarantuje, że pozostaniesz na właściwej drodze i nie wprowadzisz skomplikowanych nieuzasadnionych optymalizacji. Niezwykle istotne jest zdefiniowanie konkretnych celów i dokonanie pomiarów. Aby skutecznie przeprowadzić optymalizację, trzeba zrozumieć, jakie aspekty wydajności są ważne w danej aplikacji.

## Aspekty wydajności

Zanim zaczniesz pomiary, musisz ustalić, które aspekty wydajności są ważne w danej aplikacji. W tym podrozdziale wyjaśniamy pojęcia często używane w kontekście pomiaru wydajności. W poszczególnych aplikacjach niektóre aspekty są bardziej istotne od innych. Na przykład w internetowej usłudze do konwersji zdjęć przepustowość może być ważniejsza od latencji, natomiast latencja może być bardziej istotna w interaktywnych aplikacjach z wymogami dotyczącymi działania w czasie rzeczywistym. Oto ważne pojęcia i koncepcje, które warto znać w trakcie pomiarów wydajności:

- **Latencja i czas reakcji.** W zależności od dziedziny latencja i czas reakcji mogą mieć bardzo precyzyjnie określone i inne znaczenie. Jednak w tej książce te określenia oznaczają czas między zgłoszeniem żądania a otrzymaniem odpowiedzi, na przykład czas potrzebny do przetworzenia jednego zdjęcia w usłudze do konwersji zdjęć.
- **Przepustowość.** Oznacza liczbę transakcji (operacji, żądań itd.) przetwarzanych w jednostce czasu, na przykład liczbę zdjęć, jakie usługa do ich konwersji potrafi przetworzyć w ciągu sekundy.
- **Ograniczenie przez operacje wejścia – wyjścia lub ograniczenie przez procesor.** Przez większość czasu wykonywania zadania albo obliczane są operacje w procesorze, albo trwa oczekiwanie na operacje wejścia – wyjścia (w dysku twardym, sieci itd.). Zadanie jest ograniczone przez procesor, jeśli byłoby wykonywane szybciej z użyciem szybszego procesora. Zadanie jest ograniczone przez operacje wejścia – wyjścia, jeżeli byłoby wykonywane szybciej dzięki przyspieszeniu takich operacji. Niekiedy można zetknąć się z zadaniami ograniczonymi przez pamięć, co oznacza, że wąskim gardłem jest ilość lub szybkość pamięci roboczej.
- **Zużycie energii.** Jest to bardzo istotny aspekt dla kodu wykonywanego w urządzeniach przenośnych z zasilaniem akumulatorowym. Aby zmniejszyć zużycie energii, aplikacja musi korzystać ze sprzętu w bardziej wydajny sposób, podobnie jak optymalizowane jest używanie procesora, sieci itd. Ponadto należy unikać częstego odpytywania, ponieważ procesor nie może wtedy przejść w stan uśpienia.
- **Agregowanie danych.** Zwykle konieczna jest agregacja danych, gdy w trakcie pomiaru wydajności zbieranych jest wiele próbek. Czasem *wartość średnia* jest wystarczająco dobrym wskaźnikiem wydajności programu, jednak częściej to *mediana* daje lepsze informacje na temat rzeczywistej wydajności, ponieważ jest bardziej odporna na wartości odstające. Jeśli interesują Cię wartości odstające, zawsze możesz zmierzyć wartości *minimalną* i *maksymalną* (lub wartość 10 percentyla).

Ta lista nie jest kompletna, ale stanowi dobry punkt wyjścia. Ważne jest, aby pamiętać, że istnieją ustalone pojęcia i koncepcje związane z pomiarem wydajności. Jeśli poświęcisz trochę czasu na zdefiniowanie, co oznacza dla Ciebie optymalizacja kodu, łatwiej będzie Ci szybko osiągnąć cele.

## Przyspieszanie wykonywania kodu

Jeśli porównujesz względną wydajność dwóch wersji programu lub funkcji, zwykle celem jest **przyspieszenie** kodu. Tu przedstawiamy definicję przyspieszania związaną z czasem wykonywania kodu (lub latencją). Załóżmy, że dokonano pomiaru wykonywania dwóch wersji tego samego kodu: starszej, wolniejszej wersji i nowszego, szybszego kodu. Przyspieszenie wykonywania kodu można obliczyć w następujący sposób:

$$\text{Przyspieszenie czasu wykonywania} = \frac{T_{\text{starsza}}}{T_{\text{nowsza}}}$$

W tym wzorze  $T_{\text{starsza}}$  to czas wykonywania pierwotnej wersji kodu, a  $T_{\text{nowsza}}$  to czas wykonywania zoptymalizowanej wersji. Zgodnie z tą definicją przyspieszenie na poziomie 1 oznacza brak zmian.

Upewnijmy się teraz na podstawie przykładu, że wiesz, jak zmierzyć względny czas wykonywania kodu. Załóżmy, że funkcja jest wykonywana 10 ms ( $T_{\text{starsza}} = 10$  ms), a dzięki optymalizacji ten czas udało się skrócić do 4 ms ( $T_{\text{nowsza}} = 4$  ms). Poziom przyspieszenia można obliczyć w następujący sposób:

$$\text{Przyspieszenie} = \frac{T_{\text{starsza}}}{T_{\text{nowsza}}} = \frac{10 \text{ ms}}{4 \text{ ms}} = 2,5$$

Oznacza to, że nowa, zoptymalizowana wersja zapewnia 2,5-krotne przyspieszenie. Jeśli chcesz wyrazić tę poprawę procentowo, możesz użyć poniższego wzoru, aby przekształcić poziom przyspieszenia na procentową zmianę:

$$\text{Poprawa procentowo} = 100 \left( 1 - \frac{1}{\text{przyspieszenie}} \right) = 100 \left( 1 - \frac{1}{2,5} \right) = 60\%$$

Można stwierdzić, że nowa wersja działa o 60% szybciej niż starsza, co odpowiada 2,5-krotnemu przyspieszeniu. W tej książce w porównaniach czasu wykonywania będziemy konsekwentnie używać krotności, a nie procentowej poprawy.

Zazwyczaj interesuje nas czas wykonywania, jednak nie zawsze jego pomiar jest najlepszym rozwiązaniem. Dzięki sprawdzeniu innych wartości w sprzęcie może on zapewnić nam inne przydatne wskazówki związane z optymalizacją kodu.

## Liczniki wydajności

Oprócz oczywistych aspektów takich jak czas wykonywania i zużycie pamięci niekiedy przydatny jest pomiar także innych wartości. Mogą być one bardziej wiarygodne lub dawać lepszy wgląd w przyczyny powolnego działania kodu.

Wiele procesorów jest wyposażonych w sprzętowe liczniki wydajności, które zapewniają informacje o liczbie instrukcji, cyklach procesora, błędnych predykcjach rozgałęzień i brakach danych w pamięci podręcznej. Na razie nie wspominaliśmy o tych aspektach sprzętowych

i nie będziemy szczegółowo omawiać liczników wydajności. Warto jednak wiedzieć, że takie liczniki istnieją oraz że we wszystkich popularnych systemach operacyjnych dostępne są gotowe narzędzia i biblioteki (dostępne za pomocą API) do rejestrowania wskazań **liczników PMC** (ang. *Performance Monitoring Counter*) w trakcie wykonywania programu.

Obsługa liczników wydajności zależy od procesora i systemu operacyjnego. Intel udostępnia rozbudowane narzędzie VTune, które można zastosować do monitorowania wskaźników wydajności. W systemie FreeBSD dostępne jest narzędzie pmcstat, a w systemie macOS możesz użyć DTrace i Xcode Instruments. W środowisku Microsoft Visual Studio dostępna jest obsługa rejestrowania wskazań liczników procesora w systemie Windows.

Innym popularnym narzędziem jest perf, dostępne w systemach GNU i Linux. Uruchamia się je tak:

```
perf stat ./Twój-program
```

W ten sposób można się wiele dowiedzieć o interesujących zdarzeniach takich jak liczba przełączeń kontekstu, błędów stron, błędnych predykcji rozgałęzień itd. Oto przykładowe dane wyjściowe dla krótkiego programu:

```
Performance counter stats for './my-prog':
    1 129,86 msec task-clock          # 1,000 CPUs utilized
           8 context-switches      # 0,007 K/sec
           0 cpu-migrations         # 0,000 K/sec
        97 810 page-faults         # 0,087 M/sec
    3 968 043 041 cycles              # 3,512 GHz
    1 250 538 491 stalled-cycles-frontend # 31,52% frontend cycles idle
    497 225 466 stalled-cycles-backend  # 12,53% backend cycles idle
    6 237 037 204 instructions       # 1,57 insn per cycle
                                           # 0,20 stalled cycles per insn
    1 853 556 742 branches           # 1640,516 M/sec
           3 486 026 branch-misses  # 0,19% of all branches

    1,130355771 sec time elapsed

    1,026068000 sec user
    0,104210000 sec sys
```

Teraz przejdziemy do dobrych praktyk związanych z testami i ocenianiem wydajności.

## Testy wydajności — dobre praktyki

Z jakichś powodów w testach regresji częściej sprawdzane są wymagania funkcjonalne niż wymagania dotyczące wydajności lub innych aspektów нефункциональных. Testy wydajności przeważnie są przeprowadzane sporadycznie i zazwyczaj zbyt późno w toku prac. Zalecamy, aby mierzyć wydajność wcześniej i wykrywać regresję tak szybko, jak jest to możliwe, dodając testy wydajności do codziennego procesu kompilacji.



Jeśli kod ma obsługiwać duże dane wejściowe, starannie dobierz algorytmy i struktury danych, ale nie zajmuj się szczegółowym dostrajaniem kodu, jeżeli nie ma ku temu dobrych powodów. Ważne jest też, aby szybko przetestować aplikacje z użyciem realistycznych danych testowych. Na wczesnym etapie projektu zastanów się nad ilością danych. Ile wierszy tabeli aplikacja ma obsługiwać przy zachowaniu możliwości płynnego przewijania? Nie ograniczaj się do wypróbowania aplikacji ze 100 elementami z nadzieją, że kod będzie się skalował. Przetestuj to!

Tworzenie wykresów jest bardzo skutecznym sposobem na zrozumienie zebranych danych. Istnieje wiele dobrych i łatwych w użyciu narzędzi do tworzenia wykresów, dlatego nie masz wymówek do rezygnacji z ich generowania. RStudio i Octave zapewniają rozbudowane możliwości w zakresie tworzenia wykresów. Inne przykłady to gnuplot i Matplotlib (w Pythonie), które są dostępne w różnych systemach i umożliwiają za pomocą prostych skryptów generowanie przydatnych wykresów po zebraniu danych. Wykres nie musi być wyglądać atrakcyjnie, aby był przydatny. Po utworzeniu wykresu z danymi zobaczysz wartości odstające i wzorce, które zwykle trudno jest dostrzec w tabeli pełnej liczb.

Na tym zakończymy podrozdział „Co mierzyć i w jaki sposób?”. Teraz przejdziemy do sposobów znajdowania krytycznych fragmentów kodu, które zużywają za dużo zasobów.

## Poznaj kod i znajdź hot spoty

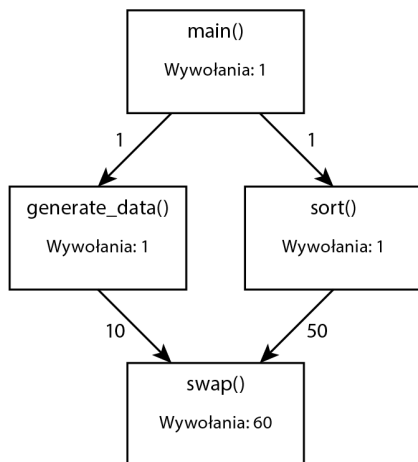
Zasada Pareto (zasada 80/20) jest stosowana w różnych dziedzinach od czasu, gdy ponad 100 lat temu została po raz pierwszy zaobserwowana przez włoskiego ekonomistę Vilfredo Pareto. Wykazał on, że 20% Włochów posiada 80% ziemi. Ta zasada jest też często używana (a może nawet nadużywana) w informatyce. W dziedzinie optymalizacji oprogramowania z tej zasady wynika, że 20% kodu odpowiada za 80% zasobów zużywanych przez program.

Jest to oczywiście tylko prosta reguła, której nie należy traktować dosłownie. Jeśli jednak kod nie jest zoptymalizowany, często można znaleźć stosunkowo niewielkie hot spoty, w których program zużywa zdecydowaną większość wszystkich zasobów. Dla programisty jest to dobra informacja, ponieważ oznacza, że można napisać większość kodu bez dopracowywania go pod kątem wydajności i zamiast tego skupić się na zachowaniu przejrzystości. Ponadto oznacza to, że w trakcie wprowadzania optymalizacji trzeba wiedzieć, *gdzie* je zastosować. W przeciwnym razie istnieje duże ryzyko, że zoptymalizowany zostanie kod, który nie ma wpływu na ogólną wydajność aplikacji. W tym podrozdziale omawiamy metody i narzędzia do znajdowania 20% kodu, który może być wart optymalizacji.

Używanie profilera jest zwykle najwydajniejszym sposobem na identyfikowanie hot spotów w programie. Profilerzy analizują wykonywanie programu i zwracają statystyczne podsumowanie (profil) określające, jak często funkcje lub instrukcje programu są wywoływane.

Ponadto profilerzy przeważnie zwracają graf wywołań, który ilustruje zależności między wywołaniami funkcji, czyli jednostki wywołujące i wywoływane dla każdej funkcji wywołanej w trakcie

profilowania. Na rysunku 3.4 widać, że funkcja `sort()` została wywołana w funkcji `main()` (jednostka wywołująca), a funkcja `sort()` wywołała funkcję `swap()` (jednostka wywoływana).



**Rysunek 3.4.** Przykładowy graf wywołań. Funkcja `sort()` jest wywoływana raz i wywołuje funkcję `swap()` 50 razy

Istnieją dwie główne kategorie profilerów: profilerzy z próbkowaniem i profilerzy z instrumentacją. Te podejścia można łączyć, tworząc hybrydowe profilerzy z próbkowaniem i instrumentacją. Tak działa na przykład `gprof`, uniksowe narzędzie do analizy wydajności. W dalszych punktach skupiamy się na profilerach z instrumentacją i profilerach z próbkowaniem.

## Profilerzy z instrumentacją

Instrumentacja polega na wstawianiu kodu do analizowanego programu w celu zbierania informacji o tym, jak często każda funkcja jest wykonywana. Wstawiony kod do instrumentacji zwykle rejestruje wszystkie punkty wejścia i wyjścia. Możesz napisać własny prosty profiler z instrumentacją, ręcznie dodając potrzebny kod. Możesz też użyć narzędzia, które automatycznie wstawia niezbędny kod na etapie procesu budowania.

W danych okolicznościach prosta implementacja może być wystarczająco dobra, należy jednak pamiętać o wpływie dodanego kodu na wydajność, przez co uzyskany profil może być mylący. Innym problemem z prostymi implementacjami jest to, że mogą one uniemożliwić wprowadzenie optymalizacji przez kompilator lub spowodować usunięcie sprawdzanego kodu w wyniku optymalizacji.

Aby przedstawić przykład profilerzy z instrumentacją, pokazujemy uproszczoną wersję klasy zegara, której jeden z autorów używał we wcześniejszych projektach:

```

class ScopedTimer {
public:
    using ClockType = std::chrono::steady_clock;

```

```

ScopedTimer

r(const char* func)
    : function_name_{func}, start_{ClockType::now()} {}

ScopedTimer(const ScopedTimer&) = delete;
ScopedTimer(ScopedTimer&&) = delete;
auto operator=(const ScopedTimer&) -> ScopedTimer& = delete;
auto operator=(ScopedTimer&&) -> ScopedTimer& = delete;

~ScopedTimer() { using namespace std::chrono;
    auto stop = ClockType::now();
    auto duration = (stop - start_);
    auto ms = duration_cast<milliseconds>(duration).count();
    std::cout << ms << " ms " << function_name_ << '\n';
}

private:
    const char* function_name_;
    const ClockType::time_point start_;
};

```

Klasa `ScopedTimer` mierzy czas od momentu utworzenia zegara do chwili jego wyjścia z zasięgu (czyli usunięcia). Używamy tu dostępnej od standardu C++11 klasy `std::chrono::steady_clock`, która została zaprojektowana do pomiaru przedziałów czasu. Klasa `steady_clock` działa monotonicznie, co oznacza, że nigdy nie zmniejsza wartości między dwoma kolejnymi wywołaniami `clock_type::now()`. Zegar systemowy tego nie zapewnia, ponieważ może zostać przestawiony w dowolnym momencie.

Teraz możemy użyć klasy zegara i zmierzyć czas działania każdej funkcji w programie, tworząc obiekt typu `ScopedTimer` na początku wszystkich funkcji:

```

auto some_function() {
    ScopedTimer timer{"some_function"};
    // ...
}

```

Choć ogólnie nie zalecamy stosowania makr preprocesora, jest to jedna z sytuacji, gdy mogą być one przydatne:

```

#if USE_TIMER
#define MEASURE_FUNCTION() ScopedTimer timer{__func__}
#else
#define MEASURE_FUNCTION()
#endif

```

Aby pobrać nazwę funkcji, używamy tu lokalnej dla funkcji predefiniowanej zmiennej `__func__`, dostępnej od standardu C++11. W standardzie C++20 wprowadzono też przydatną klasę `std::source_location`, która udostępnia funkcje `function_name()`, `file_name()`, `line()` i `column()`. Jeśli korzystasz z kompilatora, który jeszcze nie obsługuje klasy `std::source_location`, dostępne są inne, niestandardowe predefiniowane makra, które są powszechnie obsługiwane i mogą być bardzo użyteczne w trakcie debugowania, na przykład `__FUNCTION__`, `__FILE__` i `__LINE__`.

Teraz klasę `ScopedTimer` można zastosować w następujący sposób:

```
auto some_function() {
    MEASURE_FUNCTION();
    // ...
}
```

Przy założeniu, że w trakcie kompilowania zegara zdefiniowany został symbol `USE_TIMER`, ten kod po każdym zwróceniu sterowania przez funkcję `some_function()` wygeneruje następujące dane wyjściowe:

```
2.3 ms some_function
```

Pokazaliśmy, jak można ręcznie przeprowadzić instrumentację, wstawiając kod, który wyświetla czas, jaki upłynął między dwoma miejscami w programie. Choć w niektórych scenariuszach jest to przydatne narzędzie, warto pamiętać, że tego rodzaju proste rozwiązanie może dawać mylące wyniki. W następnym punkcie omawiamy metodę profilowania, która nie wymaga żadnych modyfikacji w wykonywanym kodzie.

## Profilery z próbkowaniem

Narzędzia tego rodzaju potrafią utworzyć profil, analizując stan działającego programu w równych odstępach czasu, zwykle co 10 ms. Profilerzy z próbkowaniem zazwyczaj mają minimalny wpływ na wydajność programu i umożliwiają kompilację programu w trybie produkcyjnym, z włączonymi wszystkimi optymalizacjami. Wadą profilerów z próbkowaniem jest brak precyzji i podejście statystyczne, które jednak przeważnie nie stanowi problemu, o ile masz świadomość, że jest ono stosowane.

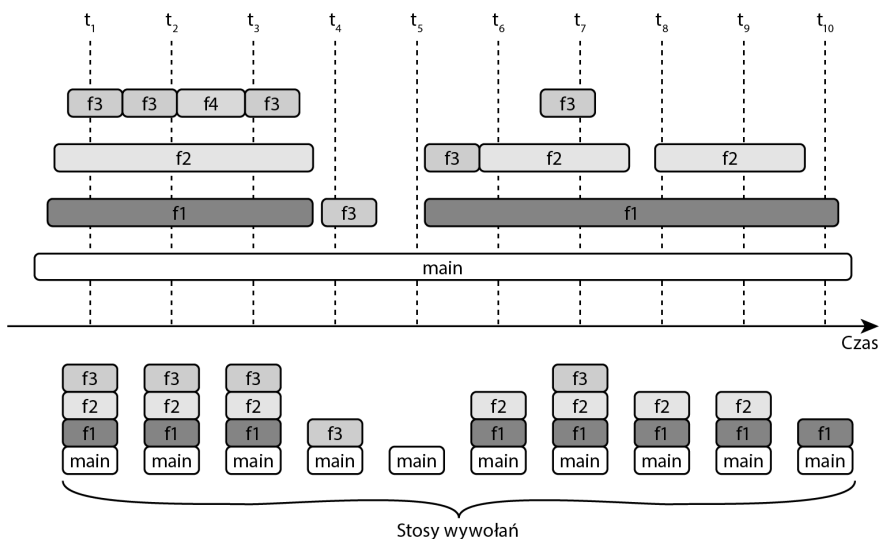
Na rysunku 3.5 pokazana jest sesja profilera z próbkowaniem dla programu mającego pięć funkcji: `main()`, `f1()`, `f2()`, `f3()` i `f4()`. Etykiety od  $t_1$  do  $t_{10}$  pokazują, kiedy pobrane zostały poszczególne próbki. Pola pokazują punkty wejścia i wyjścia każdej wykonywanej funkcji.

Podsumowanie profilu jest przedstawione w tabeli 3.3.

**Tabela 3.3.** Dla każdej funkcji profil określa procent wszystkich stosów wywołań, w których dana funkcja występuje (procent wszystkich), a także procent stosów wywołań, w których dana funkcja znajduje się w wierzchołku

| Funkcja             | Procent wszystkich | Wierzchołek |
|---------------------|--------------------|-------------|
| <code>main()</code> | 100%               | 10%         |
| <code>f1()</code>   | 80%                | 10%         |
| <code>f2()</code>   | 70%                | 30%         |
| <code>f3()</code>   | 50%                | 50%         |

Kolumna **Procent wszystkich** w tabeli 3.3 określa procent stosów wywołań, które zawierają daną funkcję. W tym przykładzie funkcja `main` występuje we wszystkich 10 z 10 stosów wywołań (100%), a funkcja `f2()` w 7 stosach (70% wszystkich stosów wywołań).



Rysunek 3.5. Przykładowa sesja profilera z próbkowaniem

Kolumna **Wierzchołek** pokazuje, ile razy każda funkcja występuje w wierzchołkach stosów wywołań. Funkcja `main()` znajduje się w wierzchołku stosu wywołań piątej próbki,  $t_5$ , a funkcja `f2()` występuje w wierzchołku stosu próbek  $t_6$ ,  $t_8$  i  $t_9$ , co odpowiada  $3/10 = 30\%$ .

Funkcja `f3()` ma najwyższą wartość w kolumnie **Wierzchołek** ( $5/10$ ) i znajdowała się w wierzchołku stosu za każdym razem, gdy została wykryta.

Koncepcyjnie profiler z próbkowaniem rejestruje próbki stosu wywołań w równych odstępach czasu. Taki profiler wykrywa, co aktualnie jest wykonywane w procesorze. Czyste profilery z próbkowaniem zwykle wykrywają tylko funkcje wykonywane w wątku, który znajduje się w stanie aktywnym, ponieważ uśpione wątki nie są szeregowane w procesorze. To oznacza, że jeśli funkcja oczekuje na blokadę, która spowodowała uśpienie wątku, ten czas oczekiwania nie zostanie uwzględniony w profilu czasowym. Jest to istotne, ponieważ wąskie gardła mogą być spowodowane synchronizacją wątków, która może być niewidoczna dla profilera z próbkowaniem.

Co się stało z funkcją `f4()`? Zgodnie z grafem została ona wywołana przez funkcję `f2()` między próbkami drugą i trzecią, jednak nie pojawiła się w profilu statystycznym, ponieważ nie została zarejestrowana w żadnym stosie wywołań. Jest to ważna cecha profilerów z próbkowaniem. Jeśli czas między rejestrowaniem próbek jest zbyt długi lub cała sesja próbkowania jest za krótka, wtedy krótkie i rzadko wywoływane funkcje nie pojawiają się w profilu. Przeważnie nie stanowi to problemu, gdyż takie funkcje rzadko wymagają dostrojenia. Można też zauważyć, że funkcja `f3()` została zagubiona między próbkami  $t_5$  i  $t_6$ , jednak ponieważ jest ona wywoływana bardzo często, i tak odgrywa ważną rolę w profilu.

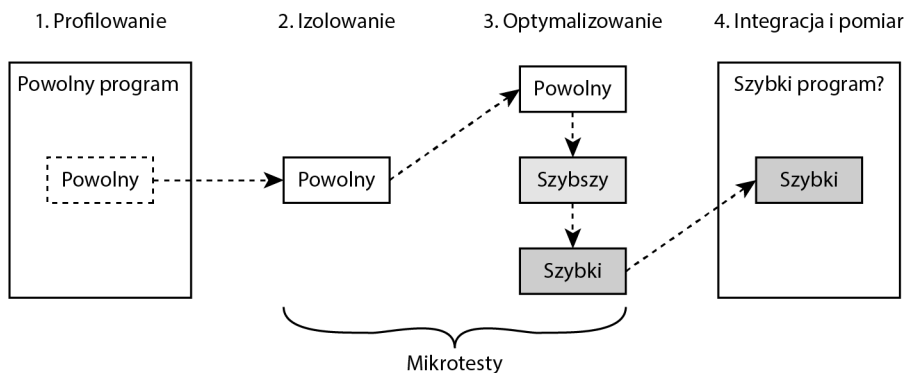
Należy koniecznie zrozumieć, co dokładnie rejestruje profiler czasu. Zwróć uwagę na jego ograniczenia i zalety, aby móc go wykorzystać tak skutecznie, jak to możliwe.

# Mikrotesty

Profilowanie pomaga znaleźć wąskie gardła w kodzie. Jeśli są one wynikiem zastosowania niewydajnych struktur danych (zobacz rozdział 4., „Struktury danych”), użycia niewłaściwych algorytmów (zobacz rozdział 5., „Algorytmy”) lub niepotrzebnej rywalizacji o zasoby (zobacz rozdział 11., „Współbieżność”), najpierw należy zająć się tego rodzaju poważniejszymi problemami. Jednak czasem natrafisz na niewielką funkcję lub krótki blok kodu, które wymagają optymalizacji. Można wtedy zastosować mikrotesty. **Mikrotest** to program, który uruchamia krótki fragment kodu izolowany od reszty aplikacji. Proces przeprowadzania mikrotestów obejmuje następujące kroki:

1. Znajdź (najlepiej za pomocą profilera) hot spot, który wymaga dostrojenia.
2. Wyodrębnij hot spot od reszty kodu i przygotuj izolowany mikrotest.
3. Zoptymalizuj kod na podstawie wyników mikrotestu. Użyj platformy testowej do testowania i oceny kodu w trakcie optymalizacji.
4. Zintegruj nowo zoptymalizowany kod z programem i *ponownie dokonaj pomiarów*, aby sprawdzić, czy optymalizacje są skuteczne, gdy kod działa w szerszym kontekście z bardziej realistycznymi danymi wejściowymi.

Te cztery kroki procesu są pokazane na rysunku 3.6.



**Rysunek 3.6.** Proces przeprowadzania mikrotestów

Przeprowadzanie mikrotestów jest dobrą zabawą. Zanim jednak spróbujesz przyspieszyć działanie konkretnej funkcji, najpierw warto sprawdzić, czy:

- czas spędzany w funkcji w trakcie wykonywania kodu ma istotny wpływ na ogólną wydajność programu, który chcesz przyspieszyć — w ocenie tego pomaga profilowanie i opisane dalej prawo Amdahla;
- nie da się w łatwy sposób zmniejszyć liczby wywołań danej funkcji — wyeliminowanie wywołań kosztownych funkcji jest przeważnie najskuteczniejszym sposobem na zoptymalizowanie ogólnej wydajności programu.

Optymalizowanie kodu na podstawie mikrotestów należy zwykle traktować jako ostateczność. Ta technika zazwyczaj daje tylko niewielki wzrost wydajności. Jednak czasem nie da się uniknąć przyspieszenia działania stosunkowo krótkiego fragmentu kodu przez modyfikację jego implementacji. W takich sytuacjach mikrotesty mogą być bardzo skuteczne.

Teraz zobaczysz, w jaki sposób przyspieszenie kodu sprawdzanego za pomocą mikrotestów pozwala przyspieszyć pracę całego programu.

## Prawo Amdahla

W trakcie stosowania mikrotestów trzeba pamiętać o tym, jak duży (lub mały) wpływ optymalizacja izolowanego kodu będzie miała na kompletny program. Z naszych doświadczeń wynika, że czasem łatwo jest popaść w nadmierną ekscytację z powodu ulepszenia kodu na podstawie mikrotestów, po czym okazuje się, że ogólny efekt zmian jest prawie pomijalny. Ryzyko można po części ograniczyć, stosując właściwe techniki profilowania, a także pamiętając o ogólnym wpływie optymalizacji na program.

Załóżmy, że na podstawie mikrotestów optymalizujemy odizolowany fragment programu. Górny limit łącznego przyspieszenia całego programu można obliczyć za pomocą prawa Amdahla. Do obliczenia ogólnego przyspieszenia potrzebne są dwie wartości:

- Po pierwsze trzeba ustalić, za jak dużą część łącznego czasu wykonywania programu odpowiada odizolowany fragment. Tę wartość oznaczymy literą  $p$  (od *proporcjonalny czas wykonywania*).
- Po drugie należy określić przyspieszenie optymalizowanego fragmentu związanego z mikrotestem. Tę wartość oznaczymy literą  $l$  (od *lokalne przyspieszenie*).

Na podstawie wartości  $p$  i  $l$  można zastosować prawo Amdahla do obliczenia ogólnego przyspieszenia programu:

$$\text{Ogólne przyspieszenie} = \frac{1}{(1 - p) + \frac{p}{l}}$$

Mamy nadzieję, że nie wydaje się to nazbyt skomplikowane. W praktyce te obliczenia okazują się łatwe do opanowania. Aby zrozumieć prawo Amdahla, warto przyrzeć się ogólnemu przyspieszeniu programu dla różnych skrajnych wartości  $p$  i  $l$ :

- Wartości  $p = 0$  i  $l = 5\times$  oznaczają, że optymalizowany fragment nie ma wpływu na ogólny czas wykonywania programu. Dlatego ogólne przyspieszenie niezależnie od wartości  $l$  zawsze będzie wynosić  $1\times$ .
- Wartości  $p = 1$  i  $l = 5\times$  oznaczają, że optymalizowany jest fragment odpowiedzialny za cały czas wykonywania programu. W tym scenariuszu ogólne przyspieszenie zawsze jest równe przyspieszeniu osiąganemu w optymalizowanym fragmencie. Tu jest to  $5\times$ .

- Wartości  $p = 0,5$  i  $l = \infty$  oznaczają, że całkowicie wyeliminowaliśmy fragment odpowiedzialny za połowę czasu wykonywania programu. Ogólne przyspieszenie wyniesie tu  $2\times$ .

Podsumowanie wyników znajduje się w tabeli 3.4.

**Tabela 3.4.** Skrajne wartości  $p$  i  $l$  oraz uzyskane ogólne przyspieszenie

| $p$ | $l$       | Ogólne przyspieszenie |
|-----|-----------|-----------------------|
| 0   | $5\times$ | $1\times$             |
| 1   | $5\times$ | $5\times$             |
| 0,5 | $\infty$  | $2\times$             |

W kompletnym przykładzie pokażemy, jak wykorzystać prawo Amdahla w praktyce. Załóżmy, że optymalizujemy funkcję w taki sposób, iż jej zoptymalizowana wersja jest 2-krotnie szybsza niż wersja pierwotna, co oznacza przyspieszenie  $2\times$  ( $l = 2$ ). Ponadto zakładamy, że ta funkcja odpowiada tylko za 1% łącznego czasu wykonywania programu ( $p = 0,01$ ). Ogólne przyspieszenie całego programu można teraz obliczyć w następujący sposób:

$$\text{Ogólne przyspieszenie} = \frac{1}{(1-p) + \frac{p}{l}} = \frac{1}{(1-0,01) + \frac{0,01}{2}} = 1,005$$

Tak więc nawet jeśli udało się 2-krotnie przyspieszyć odizolowany kod, łączne przyspieszenie wynosi 1,005 raza. Nie twierdzimy, że jest to nieistotna różnica, ale zawsze trzeba rozważać korzyści w szerszym kontekście.

## Pułapki związane z mikrotestami

Pomiary wydajności w ogóle, a zwłaszcza mikrotesty, zawsze są narażone na wiele ukrytych trudności. Oto lista kwestii, o których warto pamiętać w trakcie stosowania mikrotestów:

- Wyniki są czasem nadmiernie uogólniane i traktowane jako uniwersalna prawda.
- Kompilator może zoptymalizować odizolowany kod inaczej niż w kompletnym programie. Na przykład w mikrotestach funkcja może być rozwijana wewnątrzwerszowo, ale w kompletnym programie kompilator może nie stosować tej optymalizacji. Ponadto kompilator może przeprowadzić wstępne obliczenia dla fragmentów mikrotestu.
- Nieużywane zwracane wartości w mikrotestach mogą spowodować, że kompilator usunie funkcję, którą chcesz zmierzyć.
- Statyczne dane testowe dostępne w mikroteście mogą być dla kompilatora nierealistycznym ułatwieniem w trakcie optymalizowania kodu. Na przykład jeśli zapiszesz na stałe liczbę powtórzeń pętli, a kompilator wykryje, że ta liczba jest wielokrotnością ósemki, może przeprowadzić wektoryzację pętli w odmienny sposób, pomijając prolog i epilog dla fragmentów, które nie są wyrównane



względem wielkości rejestru SIMD. W rzeczywistym kodzie, w którym zamiast trwale zapisanej stałej z czasu kompilacji używana jest wartość z czasu wykonywania programu, taka optymalizacja nie jest wprowadzana.

- Nierealistyczne dane testowe mogą wpływać na predykcje rozgałęzień w trakcie przeprowadzania mikrotestów.
- Wyniki różnych pomiarów mogą być inne z powodu czynników takich jak skalowanie częstotliwości, zanieczyszczenie pamięci podręcznej i szeregowanie innych procesów.
- Czynnikiem ograniczającym wydajność kodu może być liczba braków danych w pamięci podręcznej, a nie czas potrzebny na wykonywanie instrukcji. Dlatego w wielu sytuacjach ważną zasadą w trakcie mikrotestów jest opróżnianie pamięci podręcznej przed pomiarami. W przeciwnym razie pomiary mogą być niewiarygodne.

Chcielibyśmy mieć prosty wzór na uniknięcie wszystkich wymienionych tu pułapek, ale niestety nie znamy go. Jednak w następnym punkcie omawiamy konkretny przykład, który ilustruje, jak można poradzić sobie z niektórymi z tych pułapek za pomocą biblioteki do przeprowadzania mikrotestów.

## Przykład ilustrujący mikrotesty

Na zakończenie rozdziału wracamy do początkowych przykładów wyszukiwania liniowego i binarnego z tego rozdziału. Pokazujemy tu, jak zbadać algorytmy za pomocą platformy do przeprowadzania mikrotestów.

Rozdział zaczęliśmy od porównania dwóch sposobów wyszukiwania liczby całkowitej w kolekcji typu `std::vector`. Jeśli wiadomo, że wektor jest już posortowany, można zastosować wyszukiwanie binarne, które jest szybsze od algorytmu wyszukiwania liniowego. Nie będziemy ponownie przedstawiać definicji obu funkcji, jednak ich deklaracje wyglądały tak:

```
bool linear_search(const std::vector<int>& v, int key);
bool binary_search(const std::vector<int>& v, int key);
```

Dla odpowiednio dużych danych wejściowych różnica w czasie wykonywania tych funkcji jest oczywista, ale w tym przykładzie są one wystarczająco dobre. Najpierw zmierzmy tylko funkcję `linear_search()`. Następnie, po przygotowaniu działającego testu, dodamy funkcję `binary_search()` i porównamy obie wersje.

Aby przygotować program testowy, najpierw potrzebujemy sposobu na wygenerowanie posortowanego wektora liczb całkowitych. Na potrzeby tego przykładu wystarczy prosta implementacja:

```
auto gen_vec(int n) {
    std::vector<int> v;
    for (int i = 0; i < n; ++i) {
        v.push_back(i);
    }
    return v;
}
```

Zwracany wektor zawiera wszystkie liczby całkowite od 0 do  $n - 1$ . Następnie możemy utworzyć „naiwny” program testowy:

```
int main() { // Nie przeprowadzaj testów wydajności w ten sposób!
    ScopedTimer timer("linear_search");
    int n = 1024;
    auto v = gen_vec(n);
    linear_search(v, n);
}
```

Szukana jest tu wartość  $n$ , o której wiadomo, że nie znajduje się w wektorze. Dlatego te dane testowe są dla algorytmu przypadkiem pesymistycznym. Jest to dobry aspekt tego testu. Oprócz tego ten test ma wiele wad, które sprawiają, że jest bezużyteczny:

- Skompilowanie tego kodu z optymalizacjami prawdopodobnie spowoduje całkowite jego usunięcie, ponieważ kompilator potrafi wykryć, że wyniki funkcji nie są używane.
- Nie chcemy mierzyć czasu potrzebnego na utworzenie i zapełnienie kolekcji typu `std::vector`.
- Tylko jednokrotne uruchomienie funkcji `linear_search()` nie pozwala uzyskać statystycznie stabilnych wyników.
- Testy dla danych wejściowych o różnej wielkości są niewygodne.

Zobacz teraz, jak można rozwiązać te problemy za pomocą biblioteki do przeprowadzania mikrotestów. Dostępne są różne narzędzia i biblioteki do przeprowadzania testów. Tu zastosujemy narzędzie **Google Benchmark** (<https://github.com/google/benchmark>), ponieważ jest często używane i ponadto umożliwia łatwe przeprowadzanie testów w internecie na stronie <http://quick-bench.com> bez konieczności instalowania czegokolwiek.

Oto jak mogą wyglądać proste mikrotesty funkcji `linear_search()` z użyciem narzędzia Google Benchmark:

```
#include <benchmark/benchmark.h> // Niestandardowy plik nagłówkowy
#include <vector>

bool linear_search(const std::vector<int>& v, int key) { /* ... */ }
auto gen_vec(int n) { /* ... */ }

static void bm_linear_search(benchmark::State& state) {
    auto n = 1024;
    auto v = gen_vec(n);
    for (auto _ : state) {
        benchmark::DoNotOptimize(linear_search(v, n));
    }
}

BENCHMARK(bm_linear_search); // Rejestrowanie funkcji testowej
BENCHMARK_MAIN();
```

I to tyle! Jedyny problem, jakiego jeszcze nie rozwiązaliśmy, to zapisanie na stałe wielkości danych wejściowych: 1024. Niedługo to poprawimy. Gdy skompilujesz i uruchomisz ten program, otrzymasz następujące dane wyjściowe:

| Benchmark        | Time   | CPU    | Iterations |
|------------------|--------|--------|------------|
| bm_linear_search | 361 ns | 361 ns | 1945664    |

Liczba iteracji podana w prawej kolumnie informuje, ile razy pętla musi zostać wykonana, aby uzyskane zostały statystycznie stabilne wyniki. Obiekt `state` przekazywany do funkcji testowej określa, kiedy należy zakończyć pracę. Średni czas iteracji jest podawany w dwóch kolumnach. W kolumnie **Time** określony jest czas zegarowy, w kolumnie **CPU** — czas używania procesora przez wątek główny. W tym przykładzie obie te wartości są identyczne, gdyby jednak funkcja `linear_search()` została zablokowana w oczekiwaniu na operacje wejścia – wyjścia, czas używania procesora byłby krótszy niż czas zegarowy.

Inną ważną kwestią jest to, że w rejestrowanym czasie nie jest uwzględniany czas pracy kodu, który generuje wektor. Jedynym mierzonym kodem jest ten, który działa w pętli:

```
for (auto _ : state) { // Pomiar dotyczy tylko tej pętli
    benchmark::DoNotOptimize(binary_search(v, n));
}
```

Wartość logiczna zwracana przez funkcje wyszukiwania jest umieszczana w wywołaniu `benchmark::DoNotOptimize()`. Ten mechanizm gwarantuje, że optymalizacje nie spowodują usunięcia zwracanej wartości, przez co całe wywołanie funkcji `linear_search()` zostałoby pominięte.

Teraz sprawimy, że testy staną się bardziej interesujące. W tym celu będziemy zmieniać wielkość danych wejściowych. Można to zrobić, przekazując argumenty do funkcji testowej za pomocą obiektu `state`:

```
static void bm_linear_search(benchmark::State& state) {
    auto n = state.range(0);
    auto v = gen_vec(n);
    for (auto _ : state) {
        benchmark::DoNotOptimize(linear_search(v, n));
    }
}
BENCHMARK(bm_linear_search)->RangeMultiplier(2)->Range(64, 256);
```

Ten test zaczyna się od danych wejściowych o wielkości 64 i podwaja tę wartość do momentu osiągnięcia rozmiaru 256. Na komputerze jednego z autorów ten test dał następujące dane wyjściowe:

| Benchmark            | Time    | CPU     | Iterations |
|----------------------|---------|---------|------------|
| bm_linear_search/64  | 17.9 ns | 17.9 ns | 38143169   |
| bm_linear_search/128 | 44.3 ns | 44.2 ns | 15521161   |
| bm_linear_search/256 | 74.8 ns | 74.7 ns | 8836955    |

W ramach ostatniego przykladu przetestujemy funkcje `linear_search()` i `binary_search()`, uzywajac danych wejsciowych o zmiennej dlugosci. Ponadto pozwolimy narzedziu sprbowaac oszacowac zlozoność czasow tych funkcji. W tym celu nalezy przekazac wielkość danych wejsciowych do obiektu `state` za pomoca funkcji `SetComplexityN()`. Kompletny mikrotest wyglada tak:

```
#include <benchmark/benchmark.h>
#include <vector>

bool linear_search(const std::vector<int>& v, int key) { /* ... */ }
bool binary_search(const std::vector<int>& v, int key) { /* ... */ }
auto gen_vec(int n) { /* ... */ }

static void bm_linear_search(benchmark::State& state) {
    auto n = state.range(0);
    auto v = gen_vec(n);
    for (auto _ : state) {
        benchmark::DoNotOptimize(linear_search(v, n));
    }
    state.SetComplexityN(n);
}

static void bm_binary_search(benchmark::State& state) {
    auto n = state.range(0);
    auto v = gen_vec(n);
    for (auto _ : state) {
        benchmark::DoNotOptimize(binary_search(v, n));
    }
    state.SetComplexityN(n);
}

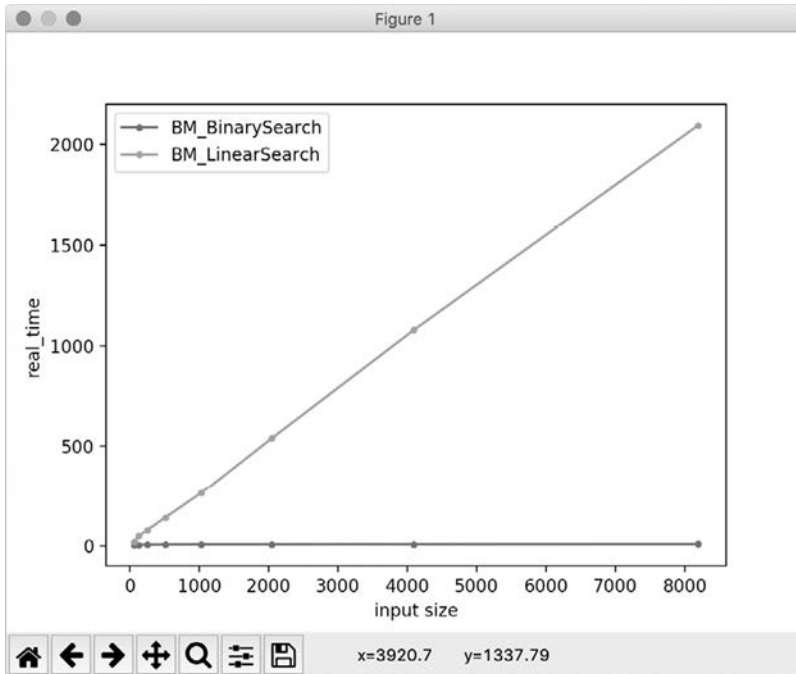
BENCHMARK(bm_linear_search)->RangeMultiplier(2)->
    Range(64, 4096)->Complexity();
BENCHMARK(bm_binary_search)->RangeMultiplier(2)->
    Range(64, 4096)->Complexity();
BENCHMARK_MAIN();
```

Gdy uruchomisz ten test, w konsoli wyswietlone zostana nastepujace wyniki:

| Benchmark                    | Time            | CPU             | Iterations |
|------------------------------|-----------------|-----------------|------------|
| bm_linear_search/64          | 18.0 ns         | 18.0 ns         | 38984922   |
| bm_linear_search/128         | 45.8 ns         | 45.8 ns         | 15383123   |
| ...                          |                 |                 |            |
| bm_linear_search/8192        | 1988 ns         | 1982 ns         | 331870     |
| <b>bm_linear_search_Big0</b> | <b>0.24 N</b>   | <b>0.24 N</b>   |            |
| bm_linear_search_RMS         | 4 %             | 4 %             |            |
| bm_binary_search/64          | 4.16 ns         | 4.15 ns         | 169294398  |
| bm_binary_search/128         | 4.52 ns         | 4.52 ns         | 152284319  |
| ...                          |                 |                 |            |
| bm_binary_search/4096        | 8.27 ns         | 8.26 ns         | 80634189   |
| bm_binary_search/8192        | 8.90 ns         | 8.90 ns         | 77544824   |
| <b>bm_binary_search_Big0</b> | <b>0.67 lgN</b> | <b>0.67 lgN</b> |            |
| bm_binary_search_RMS         | 3 %             | 3 %             |            |

Te dane wyjściowe są zgodne z wynikami z początku rozdziału, gdzie stwierdziliśmy, że algorytmy mają złożoność liniową i logarytmiczną. Jeśli wartości z tabeli przedstawimy na wykresie, można będzie łatwo dostrzec liniowe i logarytmiczne tempo wzrostu funkcji.

Rysunek 3.7 został wygenerowany za pomocą Pythona i biblioteki Matplotlib.



**Rysunek 3.7.** Wykres z czasem wykonywania funkcji dla danych wejściowych o różnej wielkości pokazuje tempo wzrostu funkcji wyszukiwania

Masz już wiele narzędzi i informacji potrzebnych do oceny i poprawy wydajności kodu. Nie sposób wystarczająco podkreślić znaczenia pomiarów i wyznaczania celów w trakcie pracy nad wydajnością. Ten podrozdział kończymy cytatem Andreia Alexandrescu:

*„Pomiary zapewniają Ci przewagę nad ekspertami, którzy nie czują potrzeby ich przeprowadzania”.*

— Andrei Alexandrescu, 2015, *Writing Fast Code I*, konferencja *code::dive 2015*,  
<https://codedive.pl/2015/writing-fast-code-part-1>

## Podsumowanie

W tym rozdziale pokazaliśmy, jak porównywać wydajność algorytmów za pomocą notacji dużego O. Wiesz już, że biblioteka standardowa języka C++ zapewnia gwarancje złożoności dla algorytmów i struktur danych. Dla wszystkich algorytmów z biblioteki standardowej określone są gwarancje wydajności dla przypadków pesymistycznego i średniego, natomiast dla kontenerów i iteratorów podana jest zamortyzowana lub dokładna złożoność.

Wiesz już też, jak ilościowo oceniać wydajność na podstawie pomiaru latencji i przepustowości.

Na zakończenie pokazaliśmy, jak wykrywać hot spoty w kodzie, używając profilerów procesora, a także jak przeprowadzać mikrotesty w celu ulepszenia izolowanych fragmentów programu.

Z następnego rozdziału dowiesz się, jak wydajnie korzystać ze struktur danych dostępnych w bibliotece standardowej języka C++.

# PROGRAM PARTNERSKI

— GRUPY 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  
**Helion** 

## Twórz wydajny i czysty kod w C++!

Dzisiejszy C++ jest wyjątkowym językiem programowania. Umożliwia pisanie zwięzłego, stabilnego kodu, który można zoptymalizować pod kątem wydajności w niespotykanym dotychczas stopniu. Język C++ w ciągu ostatnich lat został unowocześniony. W standardzie C++ 20 znalazło się sporo mechanizmów, które pozwalają osiągnąć wysoką efektywność kodu, a równocześnie uprzyjemniają programiście pracę. Poprawiono także ustawienia domyślne kompilatora. To wszystko sprawia, że wielu profesjonalistów wybiera właśnie C++, gdy chce uzyskać kod o wyjątkowej wydajności.

Ta książka jest drugim, zaktualizowanym i uzupełnionym wydaniem przewodnika dla programistów. Rozpoczyna się od szczegółowego wprowadzenia do nowoczesnego C++ z uwzględnieniem technik eliminowania wąskich gardeł w kodzie bazowym. Następnie omówiono zagadnienia optymalizacji struktur danych i zarządzania pamięcią. Przedstawiono również tematykę algorytmów, zasady pisania czytelnego kodu i stosowania niestandardowych iteratorów. Zamieszczono w niej też praktyczne przykłady używania metaprogramowania w języku C++, korutyn, refleksji (do ograniczenia ilości szablonowego kodu), obiektów pośredniczących (do wprowadzania ukrytych optymalizacji), programowania współbieżnego i struktur danych wolnych od blokad. W końcowej części dokonano przeglądu algorytmów równoległych w C++.

### W książce między innymi:

- nowe aspekty C++ 20
- wyspecjalizowane struktury danych na potrzeby wydajnego kodu
- metaprogramowanie i niestandardowe zarządzanie pamięcią
- mechanizm refleksji i programowanie współbieżne bez używania blokad
- subtelne optymalizacje algorytmów z biblioteki standardowej C++
- leniwe generatory i zadania asynchroniczne

**Björn Andrist** jest konsultantem i doświadczonym programistą C++. Tworzył kod uniksowych serwerów aplikacji oraz aplikacji audio dla komputerów stacjonarnych i urządzeń mobilnych. Prowadził kursy z zakresu algorytmów, struktur danych i programowania współbieżnego.

**Viktor Sehr** jest głównym programistą w firmie Toppluva AB produkującej gry. Od kilkunastu lat programuje w C++. Zajmował się oprogramowaniem do wizualizacji medycznej oraz aplikacjami audio.

|   |  |   |
|---|--|---|
|    | <b>KOD KORZYŚCI</b><br>Sięgnij po więcej! ▶  |  |
|  <a href="http://helion.pl">helion.pl</a>  | ISBN 978-83-283-9708-8   |   |
|  <b>HELION SA</b><br>ul. Kościuszki 1c<br>44-100 Gliwice<br>tel.: 32 250 99 63<br>helion@helion.pl | <br>9 788328 397088 |   |
| Cena: 119,00 zł   |  |   |

**Packty**