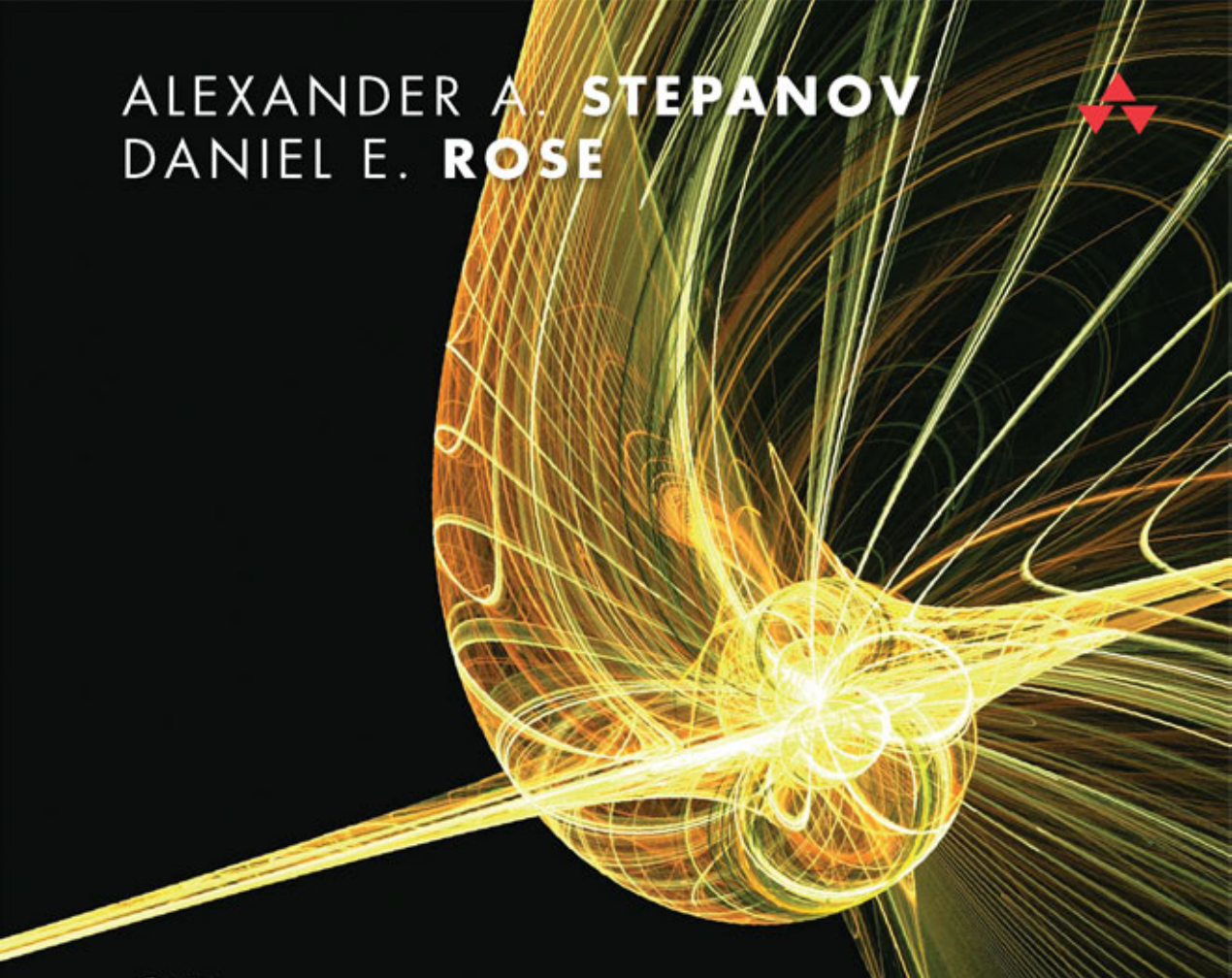
ALEXANDER A. **STEPANOV**
DANIEL E. **ROSE**



OD MATEMATYKI DO PROGRAMOWANIA UOGÓLNIONEGO

Helion

Tytuł oryginału: From Mathematics to Generic Programming

Tłumaczenie: Zdzisław Płoski

ISBN: 978-83-283-1028-5

Authorized translation from the English language edition, entitled: FROM MATHEMATICS TO GENERIC PROGRAMMING; ISBN 0321942043; by Alexander A. Stepanov; and by Daniel E. Rose; published by Pearson Education, Inc, publishing as Addison Wesley.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

Printed in Poland.

Spis treści

2 drugi

Pierwszy algorytm

Możesz szybko uczył się arytmetyki i geometrii.
[...] Wiedzę tę czerpał od Egipcjan,
którzy przedkładali naukę matematyki ponad wszystko.
Filon z Aleksandrii, *Żywot Mojżesza*

Algorytm jest skończonym ciągiem kroków składających się na wykonanie zadania obliczeniowego. Algorytmy są tak blisko związane z programowaniem komputerów, że wielu ludzi znających to pojęcie przyjmuje zapewne, że idea algorytmu pochodzi z informatyki. Jednakże algorytmy istnieją od — dosłownie — tysięcy lat. W matematyce jest mnóstwo algorytmów, a niektórymi z nich posługujemy się na co dzień. Nawet sposób dodawania długich liczb, którego uczą się dzieci w szkole, jest algorytmem.

Mimo swej długiej historii pojęcie algorytmu nie istniało od zawsze; musiało zostać wynalezione. Choć nie potrafimy określić, kiedy powstały pierwsze algorytmy, wiemy, że niektóre były znane w Egipcie już przed co najmniej czterema tysiącami lat.

* * *

Cywilizacja starożytnych Egipcjan skupiała się wzdłuż Nilu, a ich rolnictwo zależało od użyźniających glebę wylewów tej rzeki. Rodziło to trudność tego rodzaju, że po każdej powodzi wszystkie oznakowania granic poszczególnych pól były zmywane przez wodę. Do mierzenia odległości Egipcjanie używali sznurów i opracowali procedury umożliwiające powracanie do stanu zapisanego w rejestrach nieruchomości i odtwarzanie ich granic. Za zadanie to odpowiadała wybrana grupa kapłanów biegłych w owych matematycznych metodach; zwano ich „rozciągaczami sznurów". W późniejszych czasach Grecy zaczęli ich nazywać **geometrami**, czyli „mierniczymi Ziemi".

Niestety, dysponujemy niewieloma zapisami dokumentującymi matematyczną wiedzę Egipcjan. Z tego okresu przetrwały tylko dwa źródła. Jednym z nich, na którym się skoncentrujemy, jest matematyczny papirus Rhinda, nazywany tak od nazwiska XIX-wiecznego

szkockiego kolekcjonera, który kupił go w Egipcie. Jest to dokument z około 1650 roku przed naszą erą napisany przez skrybę o imieniu Ahmes. Zawiera serię problemów arytmetycznych i geometrycznych oraz kilka tabel służących do obliczeń. W tym zwoju zapisano pierwszy odnotowany algorytm, metodę szybkiego mnożenia, a także drugi — do szybkiego dzielenia. Zacznijmy od przyjrzenia się algorytmowi szybkiego mnożenia, który (jak zobaczymy w dalszej części książki) aż do dzisiaj jest ważną metodą obliczeniową.

2.1. Mnożenie po egipsku

W egipskim systemie liczbowym, tak jak we wszystkich używanych przez starożytne cywilizacje, nie posługiwano się zapisem pozycyjnym i nie istniał sposób reprezentowania zera. Wskutek tego mnożenie było niezwykle trudne i tylko niewielu wyćwiczonych rachmistrzów potrafiło je wykonywać. (Wyobraźmy sobie mnożenie dużych liczb, gdy mamy do dyspozycji tylko coś w rodzaju rzymskich liczebników).

Jak definiujemy mnożenie? Nieformalnie jest to „dodawanie czegoś do siebie pewną liczbę razy". Formalnie możemy zdefiniować mnożenie, rozbijając je na dwa przypadki: mnożenie przez 1 i mnożenie przez liczbę większą od 1.

Mnożenie przez 1 definiujemy w ten sposób:

$$1a = a. \tag{2.1}$$

Teraz mamy do czynienia z przypadkiem, w którym chcemy obliczyć iloczyn liczniejszy o jeden od już obliczonego. Niektórzy czytelnicy mogą dopatrzyć się w tym postępowania indukcyjnego; później skorzystamy z tej metody bardziej formalnie.

$$(n + 1)a = na + a. \tag{2.2}$$

Jednym ze sposobów mnożenia a przez n jest n-krotne dodanie a do siebie. Jednak w wypadku dużych liczb może to być skrajnie żmudne, gdyż wymaga $n - 1$ dodawań. W C++ taki algorytm wygląda jak niżej[1]:

```
int mnożenie0(int n, int a) {
   if (n == 1) return a;
   return mnożenie0(n - 1, a) + a;
}
```

Te dwa wiersze kodu odpowiadają równaniom 2.1 i 2.2. Zarówno a, jak i n muszą być dodatnie, tak jak to było za czasów starożytnych Egipcjan.

Algorytm opisany przez Ahmesa, wśród starożytnych Greków znany jako „mnożenie egipskie", a przez wielu współczesnych autorów określany mianem „algorytmu rosyjskich chłopów"[2], zasadza się na następującym spostrzeżeniu:

[1] Dla wygody czytania stosujemy w przekładzie wersję kodu z polskimi nazwami w pełnym alfabecie, odmienianymi w tekście; po usunięciu z liter znaków diakrytycznych kod można ćwiczyć na komputerze — *przyp. tłum.*

$$\begin{aligned} 4a &= ((a+a)+a)+a \\ &= (a+a)+(a+a). \end{aligned}$$

To udoskonalenie opiera się na prawie łączności dodawania:

$$a + (b + c) = (a + b) + c.$$

Umożliwia ono jednokrotne obliczenie $a + a$ i zmniejszenie liczby dodawań.

Pomysł polega na połowieniu n i podwajaniu a, co prowadzi do konstruowania sumy wielokrotności potęg 2. W tamtych czasach w opisach algorytmów nie posługiwano się zmiennymi w rodzaju a i n; autor podałby przykład i skwitowałby go tak: „Tedy postępuj tak samo z innymi liczbami". Ahmes nie był wyjątkiem. Przedstawił algorytm, pokazując następującą tabelę mnożenia $n = 41$ przez $a = 59$:

1	✓	59
2		118
4		236
8	✓	472
16		944
32	✓	1888

Każda pozycja po lewej jest potęgą 2. Każda pozycja po prawej jest wynikiem podwojenia poprzedniej pozycji (ponieważ dodawanie czegoś do siebie jest względnie łatwe). Zaznaczone wartości odpowiadają bitom 1 w dwójkowej reprezentacji liczby 41. Tablica w gruncie rzeczy oznajmia, że

$$41 \times 59 = (1 \times 59) + (8 \times 59) + (32 \times 59),$$

gdzie każdy z iloczynów po prawej stronie może być obliczony przez podwojenie 59 właściwą liczbę razy.

W algorytmie trzeba sprawdzać, czy n jest parzyste, czy nieparzyste, możemy więc domniemywać, że Egipcjanom znane było to rozróżnienie, choć nie mamy na to bezpośredniego dowodu. Natomiast wiedzieli o tym z pewnością starożytni Grecy, który twierdzili, że nauczyli się matematyki od Egipcjan. Oto jak zdefiniowali[3] oni liczby parzyste i nieparzyste, jeśli wyrazić to w notacji współczesnej[4]: $n = \frac{n}{2} + \frac{n}{2} \Rightarrow$ parzyste(n),

[2] Wielu informatyków zaczerpnęło tę nazwę ze *Sztuki programowania* Knutha, gdzie jest podane, że podróżujący po XIX-wiecznej Rosji spotykali chłopów używających tego algorytmu. Jednakże pierwsza wzmianka o tym pochodzi z wydanej w 1911 roku książki Sir Thomasa Heatha, który tak zauważa: „Powiedziano mi, że jest dziś w użyciu sposób (niektórzy mówią, że w Rosji, lecz *nie zdołałem tego sprawdzić*) [...]".

[3] Ta definicja występuje w pracy z I wieku zatytułowanej *Wstęp do arytmetyki*, księga I, rozdział VII, autorstwa Nikomachosa z Gerazy. Pisze on: „Parzyste jest to, co można podzielić na dwie równe części bez jedności pośrodku, a nieparzyste jest to, czego nie da się podzielić na dwie równe części z powodu wyżej wspomnianego wtrącenia jedności".

[4] Symbol strzałki „⇒" czytamy jako „implikuje". Zob. dodatek A, który zawiera zestawienie notacji matematycznej stosowanej w książce.

$$n = \frac{n-1}{2} + \frac{n-1}{2} + 1 \;\Rightarrow\; \text{nieparzyste}(n).$$

My oprzemy się również na tym wymaganiu:

$$\text{nieparzyste}(n) \Rightarrow \text{połowa}(n) = \text{połowa}(n-1)^{5}.$$

Oto jak możemy wyrazić algorytm mnożenia po egipsku w języku C++:

```
int mnożenie1(int n, int a) {
   if (n == 1) return a;
   int wynik = mnożenie1(połowa(n), a + a);
   if (nieparzyste(n)) wynik = wynik + a;
   return wynik;
}
```

Funkcję `nieparzyste(x)` możemy łatwo zaimplementować, badając najmniej znaczący bit *x*, a `połowę(x)` przez przesunięcie o jeden bit w prawo:

```
bool nieparzyste(int n) { return n & 0x1; }
int połowa(int n) { return n >> 1; }
```

Ile dodawań będzie wykonanych w `mnożeniu1`? W każdym wywołaniu funkcji musimy wykonać dodawanie wskazywane przez `+` w `a + a`. Ponieważ w trakcie rekursji połowimy wartość, wywołamy funkcję log *n* razy[6]. Czasami będziemy też musieli wykonać inne dodawanie, wskazane przez `+` w `wynik + a`. Tak więc łączna liczba dodawań wyniesie

$$\#_{+}(n) = \lfloor \log n \rfloor + (\nu(n) - 1),$$

gdzie $\nu(n)$ oznacza liczbę jedynek w dwójkowej reprezentacji *n* (*licznik populacji*, inaczej *licznik pop*). Zredukowaliśmy zatem algorytm [o złożoności] $O(n)$ do $O(\log n)$.

Czy to jest algorytm optymalny? Nie zawsze. Gdybyśmy na przykład chcieli pomnożyć przez 15, poprzedni wzór dałby taki wynik:

$$\#_{+}(15) = 3 + 4 - 1 = 6.$$

A przecież przez 15 moglibyśmy pomnożyć z użyciem tylko pięciu dodawań, korzystając z następującej procedury:

```
int mnożenie_przez_15(int a) {
   int b = (a + a) + a;            // b == 3*a
   int c = b + b;                  // c == 6*a
   return (c + c) + b;             // 12*a + 3*a
}
```

Taki ciąg dodawań jest nazywany **łańcuchem dodawań**. Odkryliśmy tu optymalny łańcuch dodawań dla 15. Niemniej algorytm Ahmesa jest w większości sytuacji wystarczająco dobry.

[5] Połowa w sensie „przepołowienia" liczby naturalnej, z odrzuceniem reszty 1 — *przyp. tłum.*

[6] Zapis „log" oznacza w całej książce logarytm przy podstawie 2, chyba że określono inaczej.

Ćwiczenie 2.1. Znajdź optymalne łańcuchy dodawań dla $n < 100$.

W którymś momencie czytelnik mógł zauważyć, że niektóre z tych obliczeń dałoby się wykonać jeszcze szybciej, gdybyśmy w sytuacji gdy pierwszy argument jest większy niż drugi, odwrócili najpierw ich kolejność (na przykład znacznie łatwiej byłoby obliczyć 3×15 niż 15×3). To prawda, Egipcjanie też o tym wiedzieli. Nie będziemy jednak dodawać tu tej optymalizacji, ponieważ — jak zobaczmy w rozdziale 7 — zechcemy na koniec uogólnić nasz algorytm na przypadki, w których argumenty są różnych typów i kolejność argumentów jest istotna.

2.2. Ulepszenie algorytmu

Nasza funkcja `mnożenie1` wypada dobrze, jeżeli bierzemy pod uwagę liczbę dodawań, jednak wykonuje ona również $\lfloor \log n \rfloor$ rekurencyjnych wywołań. Ponieważ wywołania funkcji są kosztowne, chcielibyśmy przekształcić program tak, aby uniknąć tego wydatku.

Zasada, z której tu skorzystamy, brzmi następująco. *Często łatwiej jest wykonać więcej pracy niż mniej*. W szczególności zamierzamy obliczać

$$r + na,$$

gdzie r jest bieżącym wynikiem, który gromadzi częściowe iloczyny na. Inaczej mówiąc, zamierzamy wykonać **mnożenie akumulacyjne** zamiast samego mnożenia. Ta zasada okazuje się słuszna nie tylko w programowaniu, lecz również w projektowaniu sprzętu i w matematyce, gdzie często łatwiej jest udowodnić wynik ogólny niż konkretny.

Oto jak wygląda nasza funkcja `mnożenie_akumulacyjne`:

```
int mnożenie_akumulacyjne0(int r, int n, int a) {
   if (n == 1) return r + a;
   if (nieparzyste(n)) {
      return mnożenie_akumulacyjne0(r + a, połowa(n), a + a);
   } else {
       return mnożenie_akumulacyjne0(r, połowa(n), a + a);
   }
}
```

Zachowuje ona niezmiennik $r + na = r_0 + n_0 a_0$, gdzie r_0, n_0 i a_0 są początkowymi wartościami tych zmiennych.

Możemy to jeszcze ulepszyć, upraszczając rekursję. Zauważmy, że dwa rekurencyjne wywołania różnią się tylko pierwszym argumentem. Zamiast dwóch rekurencyjnych wywołań — dla przypadku parzystego i nieparzystego — po prostu zmodyfikujemy wartość r, zanim wykonamy nawrót, w taki oto sposób:

```
int mnożenie_akumulacyjne1(int r, int n, int a) {
   if (n == 1) return r + a;
   if (nieparzyste(n)) r = r + a;
   return mnożenie_akumulacyjne1(r, połowa(n), a + a);
}
```

Obecnie nasza funkcja jest **rekursją ogonową** (ang. *tail-recursive*), tzn. wywołanie rekurencyjne występuje w niej tylko w wartości zwracanej. Wkrótce zrobimy z tego użytek.

Zauważmy dwie rzeczy:

- *n* rzadko jest równe 1;
- jeśli *n* jest parzyste, nie ma sensu sprawdzać, czy jest równe 1.

Możemy zatem dwukrotnie zmniejszyć liczbę porównań z 1, sprawdzając najpierw nieparzystość:

```
int mnożenie_akumulacyjne2(int r, int n, int a) {
   if (nieparzyste(n)) {
      r = r + a;
      if (n == 1) return r;
   }
   return mnożenie_akumulacyjne2(r, połowa(n), a + a);
}
```

Niektórzy programiści sądzą, że w procesie optymalizacji wykonywanej przez kompilator tego rodzaju przekształcenia zostaną zrobione za nas, jednak nieczęsto jest to zgodne z prawdą; kompilatory nie zamieniają jednego algorytmu na inny.

To, co zrobiliśmy do tej pory, jest całkiem niezłe, lecz dążymy do całkowitego wyeliminowania rekursji, aby uniknąć nakładów związanych z wywoływaniem funkcji. Jest to łatwiejsze do osiągnięcia, jeśli rekursja w funkcji jest ściśle ogonowa.

Definicja 2.1. Procedura rekurencyjna z **rekursją ściśle ogonową** to taka, w której wszystkie ogonowe wywołania rekurencyjne są wykonane z parametrami formalnymi procedury w roli odpowiednich argumentów.

To również możemy osiągnąć w prosty sposób, przypisując pożądane wartości zmiennym, które będziemy przekazywać, zanim spowodujemy rekursję:

```
int mnożenie_akumulacyjne3(int r, int n, int a){
   if (nieparzyste(n)) {
      r = r + a;
      if (n == 1) return r;
   }
   n = połowa(n);
   a = a + a;
   return mnożenie_akumulacyjne3(r, n, a);
}
```

Teraz już łatwo to zamienić na program iteracyjny przez zastąpienie rekursji ogonowej konstrukcją `while(true)`:

```
int mnożenie_akumulacyjne4(int r, int n, int a) {
   while (true) {
      if (nieparzyste(n)) {
         r = r + a;
         if (n == 1) return r;
      }
      n = połowa(n);
```

```
      a = a + a;
   }
}
```

Korzystając z tak zoptymalizowanej funkcji mnożenia akumulacyjnego, możemy napisać nową wersję mnożenia. Ta nowa wersja będzie wywoływać naszą pomocniczą funkcję `mnożenie_akumulacyjne4`:

```
int mnożenie2(int n, int a) {
   if (n == 1) return a;
   return mnożenie_akumulacyjne4(a, n - 1, a);
}
```

Zauważmy, że pomijamy jedną iterację `mnożenia_akumulacyjnego4`, wywołując je z wynikiem już ustawionym na *a*.

Wszystko to wygląda bardzo ładnie z wyjątkiem sytuacji, gdy *n* jest potęgą 2. Pierwsze, co wykonujemy, to odjęcie 1, co oznacza, że `mnożenie_akumulacyjne4` będzie wywoływane z liczbą, która ma w rozwinięciu dwójkowym same jedynki — jest to najgorszy przypadek w naszym algorytmie. Unikniemy tego, wykonując zawczasu nieco pracy, gdy *n* jest parzysta, połowiąc ją (i podwajając *a*) dopóty, dopóki nie stanie się nieparzysta:

```
int mnożenie3(int n, int a) {
   while (!nieparzyste(n)) {
      a = a + a;
      n = połowa(n);
   }
   if (n == 1) return a;
   return mnożenie_akumulacyjne4(a, n - 1, a);
}
```

Teraz jednak zauważamy, że zmuszamy funkcję `mnożenie_akumulacyjne4` do wykonania jednego niepotrzebnego sprawdzenia nieparzystości *n*, ponieważ wywołujemy ją z liczbą parzystą. Wykonamy więc na argumentach jedno połowienie i podwojenie przed jej wywołaniem, dochodząc w ten sposób do wersji ostatecznej:

```
int mnożenie4(int n, int a) {
   while (!nieparzyste(n)) {
      a = a + a;
      n = połowa(n);
   }
   if (n == 1) return a;
   // parzyste(n−1)  ⇒  n −1 ≠ 1
   return mnożenie_akumulacyjne4(a, połowa(n - 1), a + a);
}
```

Przepisywanie kodu

Jak można było zobaczyć na przykładzie wykonanych przez nas przekształceń, przepisywanie kodu jest ważne. Nikt nie pisze dobrego kodu za pierwszym razem. Znalezienie najefektywniejszego lub ogólnego sposobu wykonania czegoś wymaga wielu iteracji. Żaden programista nie powinien mieć jednoprzebiegowej mentalności.

W czasie tego postępowania w którymś momencie mogła Ci przyjść do głowy taka myśl: „Jedna operacja więcej nie zrobi dużej różnicy". Może się jednak okazać, że Twój kod będzie wykorzystywany ponownie wiele razy przez wiele lat. (Rzeczywistość dowodzi, że prowizorki w kodzie potrafią egzystować całymi latami). Ponadto tę taniutką operację, której w tej chwili zaoszczędzasz, w przyszłej wersji kodu ktoś mógłby zastąpić bardzo kosztowną.

Inną korzyścią z dążenia do efektywności jest to, że takie postępowanie wymusza na Tobie głębsze zrozumienie problemu. Z kolei owo głębsze zrozumienie prowadzi do efektywniejszej implementacji — działa tu efekt spirali.

2.3. Przemyślenia związane z rozdziałem

Uczący się elementarnej algebry mają upraszczać wyrażenia tak długo, jak długo się da. W naszych kolejnych realizacjach algorytmu mnożenia po egipsku przeszliśmy podobny proces, przeformułowując kod w celu uczynienia go przejrzystszym i sprawniejszym. Każdy, kto programuje, musi nabrać nawyku podejmowania prób przekształcania kodu w dążeniu do otrzymania ostatecznej jego postaci.

Zobaczyliśmy, jak powstawała matematyka w starożytnym Egipcie i jak dzięki niej zyskaliśmy pierwszy znany algorytm. Nieco dalej w książce powrócimy do niego, aby go rozszerzyć. Na razie jednak przeskoczymy o tysiąc lat do przodu, żeby przyjrzeć się odkryciom matematycznym z czasów starożytnej Grecji.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

PRZEKONAJ SIĘ, JAKIE TAJEMNICE KRYJE ŚWIAT MATEMATYKI!

Program to nic innego jak ciąg poleceń realizujących zadany algorytm. A gdy mówimy o algorytmach, jesteśmy tylko o krok od matematyki! To wyjątkowo interesująca dziedzina, którą w praktyce powinien znać każdy programista. Jeżeli chcesz zrozumieć uogólnione zasady programowania oraz podstawy matematycznych abstrakcji, na których się ono opiera, to trzymasz w rękach odpowiednią publikację.

Na kolejnych stronach znajdziesz interesujące informacje na temat pierwszych algorytmów, historii zera oraz nowoczesnych teorii liczb. Po zdobyciu podstawowych wiadomości oraz poznaniu zarysu dziejów matematyki przejdziesz do zaznajamiania się z abstrakcjami, takimi jak grupy, monoidy, półgrupy. Następnie opanujesz m.in. takie zagadnienia jak wyprowadzanie algorytmu uogólnionego, struktury algebraiczne oraz sposoby organizacji wiedzy matematycznej. Sprawdzisz też, jak wyglądają najważniejsze koncepcje programowania, co to są algorytmy permutacyjne i czym zajmuje się kryptologia. Ta książka pochłonie Cię na wiele godzin!

DZIĘKI TEJ KSIĄŻCE:

- poznasz najciekawsze historie z dziedziny matematyki
- przekonasz się, jaki wpływ ma świat matematyki na świat informatyki
- zrozumiesz ciekawe teorie związane z liczbami
- zapoznasz się z podstawowymi koncepcjami programowania
- zdobędziesz solidną wiedzę matematyczną

Alexander A. Stepanov — jest autorem licznych prac o podstawach programowania. W swojej karierze programował systemy operacyjne, narzędzia, kompilatory oraz dodatkowe biblioteki. Jest laureatem nagrody Excellence in Programming, przyznawanej przez miesięcznik „Dr. Dobb's Journal", i autorem projektu standardowej biblioteki szablonów (STL) w języku C++.

Daniel E. Rose — zajmował kierownicze stanowiska w firmach: Apple, AltaVista, Xigo, Yahoo! i A9.com. W swoich badaniach skupia się na wszystkich aspektach związanych z wyszukiwaniem danych. Na Uniwersytecie Kalifornijskim w San Diego zrobił doktorat z kognitywistyki.

Addison-Wesley

Helion

36271 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-1028-5

9 788328 310285

cena: 49,00 zł

ALWAYS LEARNING

PEARSON