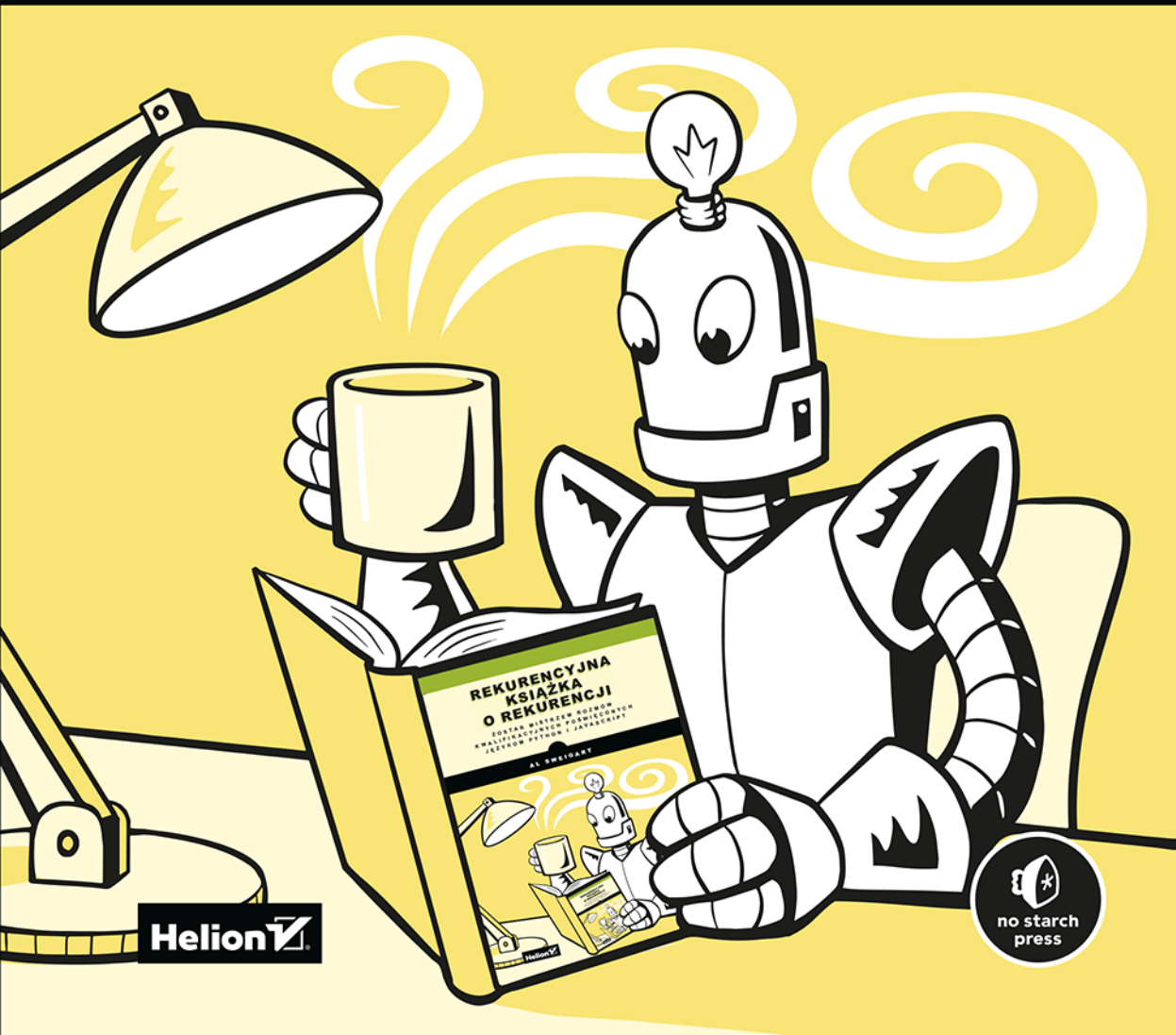
REKURENCYJNA KSIĄŻKA O REKURENCJI

ZOSTAŃ MISTRZEM ROZMÓW KWALIFIKACYJNYCH POŚWIĘCONYCH JĘZYKOM PYTHON I JAVASCRIPT

AL SWEIGART



Tytuł oryginału: The Recursive Book of Recursion: Ace the Coding Interview with Python and JavaScript

Tłumaczenie: Filip Kamiński

ISBN: 978-83-8322-653-8

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/rekuks.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

2

Rekurencja a iteracja

Nie da się jednoznacznie stwierdzić, że podejście rekurencyjne jest lepsze od iteracyjnego albo na odwrót. Praktycznie każdy kod rekurencyjny można zapisać w postaci iteracyjnej z użyciem pętli i stosu. Rekurencja nie ma żadnych specjalnych mocy pozwalających za jej pomocą wykonywać operacje niemożliwe do przeprowadzenia za pomocą algorytmów iteracyjnych. Dowolną pętlę iteracyjną można również zapisać w postaci funkcji rekurencyjnej.

W tym rozdziale porównam podejście rekurencyjne i podejście iteracyjne. Przyjrzymy się w nim klasycznym algorytmom pozwalającym obliczyć wartości kolejnych wyrazów ciągu Fibonacciego i silni oraz zobaczymy, dlaczego ich rekurencyjne wersje niespecjalnie się do czegokolwiek nadają. Na przykładzie algorytmu potęgowania przyjrzymy się też szczegółom podejścia rekurencyjnego. Rozdział ten rzuca nowe światło na rzekomą elegancję algorytmów rekurencyjnych i pokazuje, kiedy warto zastosować podejście rekurencyjne, a kiedy nie.

Obliczanie silni

Na wielu kursach informatycznych rekurencyjny algorytm obliczania silni stanowi klasyczny przykład funkcji rekurencyjnej. Silnia liczby całkowitej (nazwijmy ją n) to iloczyn wszystkich liczb całkowitych od 1 do n. Na przykład silnia z liczby 4 to $4 \cdot 3 \cdot 2 \cdot 1$, czyli 24. W matematyce silnię oznacza się symbolem wykrzyknika. Na przykład 4! oznacza **silnię z liczby 4**. W tabeli 2.1 przedstawiłem wartości silni dla pierwszych kilku liczb naturalnych.

Tabela 2.1. Silnia z pierwszych kilku liczb naturalnych

n!		Iloczyn		Wartość
1!	=	1	=	1
2!	=	1·2	=	2
3!	=	1·2·3	=	6
4!	=	1·2·3·4	=	24
5!	=	1·2·3·4·5	=	120
6!	=	1·2·3·4·5·6	=	720
7!	=	1·2·3·4·5·6·7	=	5040
8!	=	1·2·3·4·5·6·7·8	=	40 320

Silnia jest wykorzystywana w wielu rodzajach obliczeń. Można ją zastosować na przykład do znajdowania liczby permutacji. Przykładowo liczba różnych sposobów, w jakie można ustawić cztery osoby (Alicję, Bartka, Cecylię i Dawida) w kolejce jest równa 4!. Na pierwszym miejscu kolejki może stanąć jedna z czterech osób (4), trzy pozostałe osoby mogą zająć drugie miejsce (4·3), następnie dwie pozostałe osoby mogą zająć trzecie miejsce (4·3·2). Ostatnia nieumieszczona w kolejce osoba może zająć jedynie ostatnie, czwarte miejsce (4·3·2·1). Liczba sposobów, w jakie ludzie mogą być ustawieni w kolejce — to znaczy liczba permutacji — jest równa wartości silni z liczby osób.

Przyjrzyjmy się teraz iteracyjnemu i rekurencyjnemu algorytmowi obliczania silni.

Iteracyjny algorytm obliczania silni

Iteracyjnie silnię oblicza się w bardzo prosty sposób. Wystarczy w pętli pomnożyć przez siebie liczby całkowite od 1 do *n* włącznie. Algorytmy **iteracyjne** zawsze korzystają z pętli. Oto program *factorialByIteration.py*:

Python

```
def factorial(number):
    product = 1
    for i in range(1, number + 1):
        product = product * i
    return product
print(factorial(5))
```

A to odpowiadający mu skrypt *factorialByIteration.html*:

JavaScript

```
<script type="text/javascript">
function factorial(number) {
    let product = 1;
    for (let i = 1; i <= number; i++) {
        product = product * i;
    }
```

```
    return product;
}
document.write(factorial(5));
</script>
```

Po uruchomieniu tego kodu na ekranie pojawi się wartość 5!:

```
120
```

W iteracyjnym algorytmie obliczania silni nie ma nic złego — jest bardzo prosty i wykonuje swoje zadanie. Teraz przyjrzyjmy się algorytmowi rekurencyjnemu, aby lepiej poznać naturę silni i samej rekurencji.

Rekurencyjny algorytm obliczania silni

Zauważ, że silnia z 4 to 4·3·2·1, a silnia z 5 to 5·4·3·2·1. Możemy więc powiedzieć, że 5! = 5·4!. Jest to przykład *rekurencji*, ponieważ definicja silni z 5 (lub dowolnej liczby *n*) odwołuje się do definicji silni z 4 (w ogólności z *n* – 1). Z kolei 4! = 4·3! i tak dalej, aż do momentu, w którym dochodzimy do przypadku bazowego 1!. Jeden silnia to po prostu 1.

Oto program *factorialByRecursion.py*, który oblicza silnię z użyciem algorytmu rekurencyjnego:

Python

```
def factorial(number):
    if number == 1:
        # PRZYPADEK BAZOWY
        return 1
    else:
        # PRZYPADEK REKURENCYJNY
        return number * factorial(number - 1) ❶
print(factorial(5))
```

Ten sam program zapisany w JavaScripcie (*factorialByRecursion.html*) wygląda następująco:

JavaScript

```
<script type="text/javascript">
function factorial(number) {
    if (number === 1) {
        // PRZYPADEK BAZOWY
        return 1;
    } else {
        // PRZYPADEK REKURENCYJNY
        return number * factorial(number - 1); ❶
    }
}
document.write(factorial(5) + "<br />");
</script>
```

Gdy uruchomisz ten kod, aby obliczyć rekurencyjnie 5!, na ekranie zobaczysz identyczny wynik jak w przypadku programu iteracyjnego:

```
120
```

Dla wielu programistów powyższy kod wygląda dziwnie. Niby wiadomo, że w wywołaniu `factorial(5)` musimy obliczyć iloczyn 5·4·3·2·1, ale trudno jest wskazać wiersz kodu, w którym odbywa się to mnożenie.

To zdziwienie powstaje, ponieważ przypadek rekurencyjny zawiera tylko jedną linię ❶, której połowa jest wykonywana przed wywołaniem rekurencyjnym, a połowa po powrocie z wywołania rekurencyjnego. Nie jesteśmy przyzwyczajeni do tego, że jednocześnie wykonywana jest tylko połowa linii kodu.

Pierwsza połowa instrukcji to wywołanie `factorial(number - 1)`. Jej wykonanie wiąże się z obliczeniem wartości wyrażenia `number - 1` i wywołaniem rekurencyjnym, które powoduje umieszczenie nowego obiektu ramki na stosie wywołań. Obiekt jest umieszczany na stosie przed wykonaniem wywołania rekurencyjnego.

Wykonanie powróci do poprzedniego obiektu ramki dopiero po zwróceniu wartości przez wywołanie `factorial(number - 1)`. Gdy wywoływane jest `factorial(5)`, wywołaniu `factorial(number - 1)` będzie odpowiadało wywołanie `factorial(4)`, które zwróci `24`. W tym momencie wykonywana jest druga połowa linii. Wyrażenie `return number * factorial(number - 1)` ma teraz postać `return 5 * 24`. Z tego powodu `factorial(5)` zwraca `120`.

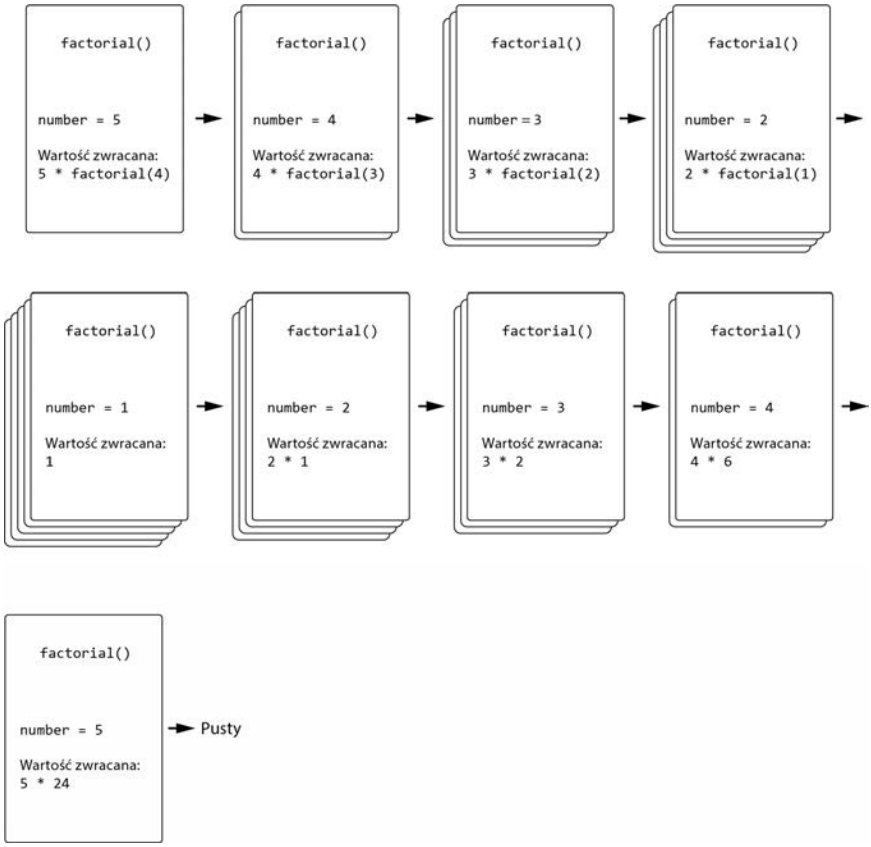
Na rysunku 2.1 pokazałem stan stosu wywołań podczas umieszczania obiektów ramek na stosie (co ma miejsce w momencie wywołania rekurencyjnego) i ich usuwania (w momencie powrotu z wywołania rekurencyjnego). Zauważ, że mnożenie następuje po wykonaniu wywołań rekurencyjnych, a nie przed.

Po zakończeniu wykonywania pierwszego wywołania funkcji `factorial()` następuję zwrócenie wartości obliczonej silni.

Dlaczego rekurencyjny algorytm obliczania silni jest szalenie nieefektywny?

Rekurencyjna implementacja algorytmu obliczania silni ma poważną wadę. Obliczenie silni z 5 wymaga pięciu wywołań rekurencyjnych. Oznacza to, że przed osiągnięciem przypadku bazowego na stos wywołań trafi pięć obiektów ramek. Zależność ta źle się skaluje.

Obliczenie silni z 1001 za pomocą rekurencyjnej funkcji `factorial()` wymaga 1001 rekurencyjnych wywołań funkcji. Wykonanie tak wielu wywołań funkcji bez powrotu prawdopodobnie zakończy się błędem przepełnienia stosu spowodowanym przekroczeniem maksymalnego rozmiaru stosu wywołań interpretera. Jest to poważna wada. W praktyce nigdy nie należy obliczać silni za pomocą funkcji rekurencyjnej.



Rysunek 2.1. Stan stosu wywołań w momencie wywoływania factorial() i powrotów z tych wywołań

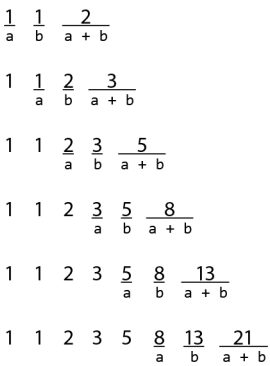
W odróżnieniu od rekurencyjnego algorytmu obliczania silni algorytm iteracyjny obliczy silnię szybko i sprawnie. W niektórych językach programowania możesz uniknąć błędu przepełnienia stosu, jeżeli zastosujesz technikę *optymalizacji rekurencji ogonowej*. Zagadnienie to omówię w rozdziale 8. Z drugiej strony technika ta dodatkowo komplikuje implementację funkcji rekurencyjnej. W przypadku obliczania silni najprostszym i najłatwiejszym do zrozumienia podejściem jest algorytm iteracyjny.

Znajdowanie wyrazów ciągu Fibonacciego

Kolejnym klasycznym przykładem prezentowanym podczas omawiania rekurencji jest **ciąg Fibonacciego**. Ciąg ten rozpoczyna się od wartości 1 i 1 (czasami spotyka się również definicję, w której pierwsze dwa wyrazy tego ciągu to 0 i 1). Każdy kolejny wyraz tego ciągu jest sumą dwóch poprzednich. Kolejne wyrazy

tego ciągu (tzw. liczby Fibonacciego) to 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 i tak dalej, aż do nieskończoności.

Oznaczmy ostatnie dwa wyrazy ciągu jako *a* i *b*. Na rysunku 2.2 pokazałem relacje pomiędzy kolejnymi elementami tego ciągu.



Rysunek 2.2. Każdy wyraz ciągu Fibonacciego jest sumą dwóch poprzednich

Przeanalizujemy teraz kilka przykładów kodu, który znajduje wyrazy ciągu Fibonacciego w iteracyjny i rekurencyjny sposób.

Iteracyjny algorytm wyznaczania n-tego wyrazu ciągu Fibonacciego

Iteracyjna implementacja algorytmu Fibonacciego jest bardzo prosta. Kod składa się z pętli `for` oraz dwóch zmiennych `a` i `b`. Na poniższym listingu pokazałem program *fibonacciByIteration.py*, zawierający iteracyjną wersję algorytmu wyznaczania wyrazów ciągu Fibonacciego zapisaną w Pythonie:

Python

```
def fibonacci(nthNumber):
    a, b = 1, 1 (1)
    print('a = %s, b = %s' % (a, b))
    for i in range(2, nthNumber):
        a, b = b, a + b # Obliczanie kolejnego wyrazu ciągu (2)
        print('a = %s, b = %s' % (a, b))
    return b

print(fibonacci(10))
```

Oto odpowiadający mu program *fibonacciByIteration.html* zapisany w Java Scripcie:

JavaScript

```
<script type="text/javascript">
function fibonacci(nthNumber) {
    let a = 1, b = 1; (1)
```

```
    let nextNum;
    document.write("a = " + a + ", b = " + b + "<br />");
    for (let i = 2; i < nthNumber; i++) {
        nextNum = a + b; // Obliczanie kolejnego wyrazu ciągu ❷
        a = b;
        b = nextNum;
        document.write("a = " + a + ", b = " + b + "<br />");
    }
    return b;
}

document.write(fibonacci(10));
</script>
```

Po uruchomieniu powyższego kodu program znajdzie 10. element ciągu Fibonacciego. Oto wyjście z tego programu:

```
a = 1, b = 1
a = 1, b = 2
a = 2, b = 3
…
a = 34, b = 55
55
```

Program musi pamiętać jedynie dwa ostatnie (w danej chwili) elementy ciągu. Ponieważ pierwsze dwa wyrazy ciągu Fibonacciego mają wartość 1, na początku programu zapisałem tę wartość w zmiennych `a` i `b` ❶. Wewnątrz pętli `for` dodaję do siebie `a` i `b` i w ten sposób obliczam kolejny wyraz ciągu ❷. Wynik tego dodawania zapisuję w zmiennej `b`. Poprzednią wartość `b` zapamiętuję zaś w zmiennej `a`. Po zakończeniu pętli zmienna `b` zawiera *n*-tą liczbę Fibonacciego, która jest następnie zwracana przez funkcję.

Rekurencyjny algorytm wyznaczania n-tego wyrazu ciągu Fibonacciego

W definicji liczb Fibonacciego ukryta jest rekurencja. Na przykład aby obliczyć 10. wyraz ciągu Fibonacciego, należy dodać do siebie 8. i 9. liczbę Fibonacciego. Aby je wyznaczyć, należy zsumować 8. i 7., a następnie 7. i 6. liczbę Fibonacciego. W obliczeniach powtarza się wiele identycznych operacji. Zauważ, że dodanie 8. liczby Fibonacciego do 9. wymaga ponownego znalezienia 8. elementu ciągu. Rekurencja kończy się po osiągnięciu przypadku bazowego, który odpowiada 1. i 2. wyrazowi ciągu, które mają wartość 1.

Oto rekurencyjna implementacja algorytmu wyznaczania *n*-tego wyrazu ciągu Fibonacciego zapisana w Pythonie (plik *fibonacciByRecursion.py*):

Python

```
def fibonacci(nthNumber):
    print('Wywołano funkcję fibonacci(%s).' % (nthNumber))
    if nthNumber == 1 or nthNumber == 2: ❶
        # PRZYPADEK BAZOWY
        print('Przypadek bazowy fibonacci(%s). Funkcja zwraca 1.' % (nthNumber))
        return 1
    else:
        # PRZYPADEK REKURENCYJNY
        print('Wywołuję fibonacci(%s) i fibonacci(%s).' %
        ↪(nthNumber - 1, nthNumber - 2))
        result = fibonacci(nthNumber - 1) + fibonacci(nthNumber - 2)
        print('Funkcja fibonacci(%s) kończy swoje działanie i zwraca
        ↪%s.' % (nthNumber, result))
        return result

print(fibonacci(10))
```

A to odpowiadająca jej implementacja w JavaScripcie (*fibonacciByRecursion.html*):

JavaScript

```
<script type="text/javascript">
function fibonacci(nthNumber) {
    document.write("Wywołano funkcję fibonacci(" + nthNumber + ").<br />");
    if (nthNumber === 1 || nthNumber === 2) { ❶
        // PRZYPADEK BAZOWY
        document.write("Przypadek bazowy fibonacci(" + nthNumber + ").
        ↪Funkcja zwraca 1.<br />");
        return 1;
    }
    else {
            // PRZYPADEK REKURENCYJNY
            document.write('Wywołuję fibonacci(' + (nthNumber - 1) + ')
            ↪i fibonacci(' + (nthNumber - 2) + ').<br />');
            let result = fibonacci(nthNumber - 1) + fibonacci(nthNumber - 2);
            document.write('Funkcja fibonacci(' + nthNumber + ') kończy swoje
            ↪działania i zwraca ' + result + '.<br />');
            return result;
    }
}

document.write(fibonacci(10) + "<br />");
</script>
```

Po uruchomieniu powyższego kodu program znajdzie 10. element ciągu Fibonacciego. Oto wyjście z tego programu:

```
Wywołano funkcję fibonacci(10).
Wywołuję fibonacci(9) i fibonacci(8).
Wywołano funkcję fibonacci(9).
Wywołuję fibonacci(8) i fibonacci(7).
Wywołano funkcję fibonacci(8).
```

```
Wywołuję fibonacci(7) i fibonacci(6).
Wywołano funkcję fibonacci(7).
…
Funkcja fibonacci(6) kończy swoje działanie i zwraca 8.
Funkcja fibonacci(8) kończy swoje działanie i zwraca 21.
Funkcja fibonacci(10) kończy swoje działanie i zwraca 55.
```

Znaczna część powyższego kodu służy jedynie do wyświetlania informacji na wyjściu. Sam sposób działania funkcji `fibonacci()` jest dość prosty. Przypadek bazowy — sytuacja, w której nie wykonujemy już kolejnych wywołań rekurencyjnych — ma miejsce, gdy `nthNumber` ma wartość `1` lub `2` ❶. W tym przypadku funkcja zwraca `1`, ponieważ pierwszy i drugi element ciągu Fibonacciego to zawsze `1`. Każdy inny przypadek jest przypadkiem rekurencyjnym, w którym zwracaną wartością jest sumą `fibonacci(nthNumber - 1)` i `fibonacci(nthNumber - 2)`. Dopóki wartość `nthNumber` jest liczbą całkowitą większą od `0`, wywołania rekurencyjne doprowadzą ostatecznie do osiągnięcia przypadku bazowego, który spowoduje zakończenie ciągu wywołań.

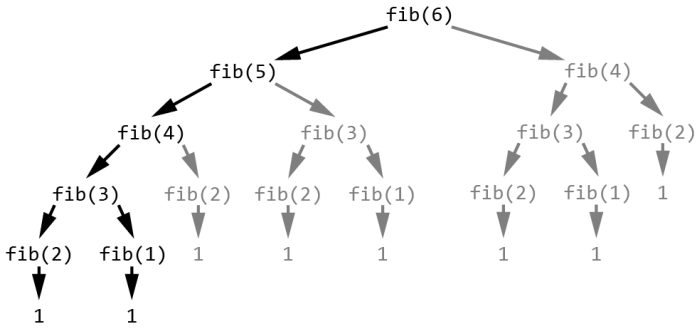
Pamiętasz, jak w rekurencyjnej implementacji silni występowała część „przed wywołaniem rekurencyjnym" i „po wywołaniu rekurencyjnym"? Ponieważ rekurencyjna implementacja algorytmu wyznaczania ciągu Fibonacciego zawiera dwa wywołania rekurencyjne w przypadku rekurencyjnym, przypadek ten dzieli się na trzy części: „przed pierwszym wywołaniem rekurencyjnym", „po pierwszym wywołaniu rekurencyjnym, ale przed drugim" oraz „po drugim wywołaniu rekurencyjnym". Obowiązują jednak te same zasady. Nie sądź, że po osiągnięciu przypadku bazowego nie ma już żadnego kodu do wykonania. Algorytm rekurencyjny kończy się dopiero po zwróceniu wartości przez pierwotne wywołanie funkcji.

Możesz zapytać: czy iteracyjna implementacja algorytmu wyznaczania n-tego wyrazu Fibonacciego nie jest prostsza niż rekurencyjna? Odpowiedź brzmi: tak. Co gorsza, rozwiązanie rekurencyjne jest mocno nieefektywne, co wyjaśnię w następnym punkcie.

Dlaczego rekurencyjny algorytm wyznaczania n-tego wyrazu ciągu Fibonacciego jest mocno nieefektywny?

Tak jak rekurencyjny algorytm obliczania silni, rekurencyjny algorytm wyznaczania *n*-tego elementu ciągu Fibonacciego ma krytyczną słabość. Otóż algorytm ten powtarza w kółko te same obliczenia. Spójrz na rysunek 2.3 i zobacz, jak w wywołaniu `fibonacci(6)` (dla zwięzłości oznaczonym jako `fib(6)`) wywołane są `fibonacci(5)` i `fibonacci(4)`.

Wywołanie funkcji powoduje kaskadę kolejnych, aż do osiągnięcia przypadków bazowych `fibonacci(2)` i `fibonacci(1)`, które zwracają `1`. Zauważ, że funkcja `fibonacci(4)` jest wywoływana dwukrotnie, `fibonacci(3)` trzykrotnie i tak dalej. Spowalnia to cały algorytm, powodując niepotrzebne powtarzanie tych samych obliczeń. Nieefektywność wzrasta, gdy szukamy dalszych elementów ciągu. Podczas gdy iteracyjny algorytm może obliczyć setny wyraz ciągu w mniej niż sekundę, algorytm rekurencyjny potrzebowałby na znalezienie tej wartości ponad miliona lat.



Rysunek 2.3. Drzewo wywołań rekurencyjnych dla wywołania fibonacci(6) (na szaro zostały zaznaczone nadmiarowe wywołania funkcji)

Zamiana algorytmu rekurencyjnego na iteracyjny

Przekształcenie algorytmu rekurencyjnego w iteracyjny jest zawsze możliwe. Funkcje rekurencyjne powtarzają obliczenia, wywołując same siebie, ale ten sam efekt można otrzymać za pomocą pętli. Funkcje rekurencyjne wykorzystują też stos wywołań, który w algorytmie iteracyjnym możemy zastąpić zwyczajnym stosem. A zatem każdy algorytm rekurencyjny może być wykonany iteracyjnie przy użyciu pętli i stosu.

Na potrzeby demonstracji spójrz na program *factorialEmulateRecursion.py*, zapisany w Pythonie. Poniższy program zawiera iteracyjny algorytm emulujący algorytm rekurencyjny:

Python

```
callStack = [] # Jawnie zdefiniowany stos wywołań, na którym są przechowywane obiekty ramek ❶
callStack.append({'returnAddr': 'start', 'number': 5}) # Wywołanie funkcji factorial() ❷
returnValue = None

while len(callStack) > 0:
    # Ciało funkcji factorial():

    number = callStack[-1]['number'] # Ustawienie wartości parametru number
    returnAddr = callStack[-1]['returnAddr']

    if returnAddr == 'start':
        if number == 1:
            # PRZYPADEK BAZOWY
            returnValue = 1
            callStack.pop() # Powrót z wywołania funkcji ❸
            continue
        else:
            # PRZYPADEK REKURENCYJNY
            callStack[-1]['returnAddr'] = 'after recursive call'
```

```
            # Wywołanie funkcji factorial():
            callStack.append({'returnAddr': 'start', 'number': number - 1}) ❹
            continue
    elif returnAddr == 'after recursive call':
        returnValue = number * returnValue
        callStack.pop() # Powrót z wywołania funkcji ❺
        continue

print(returnValue)
```

Oto odpowiadający mu program *factorialEmulateRecursion.html* zapisany w JavaScripcie:

JavaScript

```
<script type="text/javascript">
let callStack = []; // Jawnie zdefiniowany stos wywołań, na którym są przechowywane obiekty
                    // ramek ❶
callStack.push({"returnAddr": "start", "number": 5}); // Wywołanie funkcji factorial() ❷
let returnValue;

while (callStack.length > 0) {
    // Ciało funkcji factorial():
    let number = callStack[callStack.length - 1]["number"]; // Ustawienie wartości
                                                            // parametru number
    let returnAddr = callStack[callStack.length - 1]["returnAddr"];

    if (returnAddr == "start") {
        if (number === 1) {
            // PRZYPADEK BAZOWY
            returnValue = 1;
            callStack.pop(); // Powrót z wywołania funkcji ❸
            continue;
        } else {
            // PRZYPADEK REKURENCYJNY
            callStack[callStack.length - 1]["returnAddr"] = "after recursive call";
            // Wywołanie funkcji factorial():
            callStack.push({"returnAddr": "start", "number": number - 1}); ❹
            continue;
        }
    } else if (returnAddr == "after recursive call") {
        returnValue = number * returnValue;
        callStack.pop(); // Powrót z wywołania funkcji ❺
        continue;
    }
}

document.write(returnValue + "<br />");
</script>
```

Zauważ, że powyższy program nie zawiera funkcji rekurencyjnej; w ogólne nie ma w nim żadnych funkcji! Program emuluje wywołania rekurencyjne, korzystając z listy, która pełni rolę stosu naśladującego stos wywołań (zmienna `callStack`; ❶). Rolę obiektu ramki pełni słownik ❷, który przechowuje informacje o adresie zwrotnym i wartości lokalnej zmiennej `nthNumber`. Program emuluje wywołania funkcji poprzez umieszczenie obiektów ramek na stosie wywołań ❹. Zwrócenie wartości przez wywołanie jest realizowane poprzez usunięcie obiektu ramki ze stosu ❸ i ❺.

Metoda ta pozwala zapisać każdą funkcję rekurencyjną w sposób iteracyjny. Chociaż powyższy kod jest niezwykle trudny do zrozumienia (pewnie nigdy nie zapisałbyś w ten sposób algorytmu obliczania silni), pokazuje on, że rekurencja nie oferuje żadnych możliwości, których nie ma kod iteracyjny.

Zamiana algorytmu iteracyjnego na rekurencyjny

Konwersja algorytmu iteracyjnego na rekurencyjny również jest zawsze możliwa. Algorytm iteracyjny to po prostu kod wykorzystujący pętlę. Powtarzany kod (ciało pętli) można umieścić w ciele funkcji rekurencyjnej. Efekt wielokrotnego powtórzenia kodu w ciele pętli można osiągnąć za pomocą wielokrotnego wywoływania funkcji. Da się to zrobić poprzez wywołanie funkcji z poziomu niej samej (tym samym tworząc funkcję rekurencyjną).

Oto kod w Pythonie (*hello.py*), który za pomocą pętli pięciokrotnie wyświetla tekst *Witaj, świecie!*, a następnie robi to samo za pomocą funkcji rekurencyjnej:

Python

```
print('Kod w pętli:')
i = 0
while i < 5:
    print(i, 'Witaj, świecie!')
    i = i + 1

print('Kod w postaci funkcji:')
def hello(i=0):
    print(i, 'Witaj, świecie!')
    i = i + 1
    if i < 5:
        hello(i) # PRZYPADEK REKURENCYJNY
    else:
        return # PRZYPADEK BAZOWY
hello()
```

A to jego odpowiednik zapisany w JavaScripcie (*hello.html*):

JavaScript

```
<script type="text/javascript">
document.write("Kod w pętli:<br />");
let i = 0;
while (i < 5) {
    document.write(i + " Witaj, świecie!<br />");
    i = i + 1; }

document.write("Kod w postaci funkcji:<br />");
function hello(i) {
    if (i === undefined) {
        i = 0; // Jeżeli zmienna i nie ma przypisanej żadnej wartości, to przypisujemy jej 0
    }

    document.write(i + " Witaj, świecie!<br />");
    i = i + 1;
    if (i < 5) {
        hello(i); // PRZYPADEK REKURENCYJNY
    }
    else {
        return; // PRZYPADEK BAZOWY
    }
}
hello();
</script>
```

Każdy z powyższych programów wyświetli następujące dane wyjściowe:

```
Kod w pętli:
0 Witaj, świecie!
1 Witaj, świecie!
2 Witaj, świecie!
3 Witaj, świecie!
4 Witaj, świecie!
Kod w postaci funkcji:
0 Witaj, świecie!
1 Witaj, świecie!
2 Witaj, świecie!
3 Witaj, świecie!
4 Witaj, świecie!
```

Pętla `while` zawiera warunek `i < 5`, który określa, czy program ma kontynuować pętlę. Ten sam warunek jest też wykorzystywany w funkcji rekurencyjnej do ustalenia, czy należy wykonać przypadek rekurencyjny po wyświetleniu tekstu `Witaj, świecie!`, aby ponownie wyświetlić to zdanie.

Spójrz teraz na bardziej rzeczywisty przykład. Poniżej pokazałem iteracyjną i rekurencyjną funkcję zwracającą indeks początku podłańcucha `needle` w łańcuchu `haystack`. Jeśli podłańcuch nie zostanie znaleziony, obie funkcje zwrócą `-1`.

Funkcje te działają podobnie do metody `find()` łańcuchów znaków w Pythonie i metody `indexOf()` łańcuchów znaków w JavaScripcie. Oto program *findSubstring.py* zapisany w Pythonie:

Python

```
def findSubstringIterative(needle, haystack):
    i = 0
    while i < len(haystack):
        if haystack[i:i + len(needle)] == needle:
            return i # Wartość needle znaleziona
        i = i + 1
    return -1 # Nie znaleziono wartości needle

def findSubstringRecursive(needle, haystack, i=0):
    if i >= len(haystack):
        return -1 # PRZYPADEK BAZOWY (nie znaleziono wartości needle)

    if haystack[i:i + len(needle)] == needle:
        return i # PRZYPADEK BAZOWY (wartość needle znaleziona)
    else:
        # PRZYPADEK REKURENCYJNY
        return findSubstringRecursive(needle, haystack, i + 1)

print(findSubstringIterative('kot', 'Mój kot Zosia'))
print(findSubstringRecursive('kot', 'Mój kot Zosia'))
```

A to program *findSubstring.html* będący jego odpowiednikiem w JavaScripcie:

JavaScript

```
<script type="text/javascript">
function findSubstringIterative(needle, haystack) {
    let i = 0;
    while (i < haystack.length) {
        if (haystack.substring(i, i + needle.length) == needle) {
            return i; // Wartość needle znaleziona
        }
        i = i + 1
    }
    return -1; // Nie znaleziono wartości needle
}

function findSubstringRecursive(needle, haystack, i) {
    if (i === undefined) {
        i = 0;
    }

    if (i >= haystack.length) {
        return -1; // PRZYPADEK BAZOWY (nie znaleziono wartości needle)
    }

    if (haystack.substring(i, i + needle.length) == needle) {
        return i; // PRZYPADEK BAZOWY (wartość needle znaleziona)
```

```
    } else {
        // PRZYPADEK REKURENCYJNY
        return findSubstringRecursive(needle, haystack, i + 1);
    }
}

document.write(findSubstringIterative("kot", "Mój kot Zosia") + "<br />");
document.write(findSubstringRecursive("kot", "Mój kot Zosia") + "<br />");
</script>
```

Powyższe programy zawierają wywołania funkcji `findSubstringIterative()` i `findSubstring Recursive()`, które zwracają `4`, ponieważ jest to indeks, pod którym w łańcuchu `Mój kot Zosia` znajduje się słowo `kot`:

```
4
4
```

Programy z tego punktu pokazują, że zawsze możemy zamienić pętlę w równoważną jej funkcję rekurencyjną. Chociaż zastąpienie pętli rekurencją jest możliwe, odradzam takie postępowanie. Moim zdaniem jest to „rekurencja dla samej rekurencji". Ponieważ rekurencja jest często trudniejsza do zrozumienia niż kod iteracyjny, działanie takie zmniejsza czytelność kodu.

Studium przypadku: obliczanie potęg

Chociaż użycie rekurencji niekoniecznie prowadzi do powstania lepszego kodu, podejście rekurencyjne może dać nowy wgląd w problem programistyczny. W ramach studium przypadku przyjrzyjmy się, jak obliczyć potęgi.

Potęgę liczby oblicza się poprzez pomnożenie jej przez siebie samą określoną liczbę razy. Na przykład *trzy podniesione do szóstej potęgi*, czyli 3^6, jest równe sześciokrotnemu pomnożeniu liczby 3 przez samą siebie: 3·3·3·3·3·3 = 729. Operacja ta jest tak powszechna, że Python zawiera specjalny operator potęgowania `**`. W JavaScripcie potęgę obliczysz za pomocą funkcji `Math.pow()`. W Pythonie wartość 3^6 możesz wyznaczyć za pomocą instrukcji `3 ** 6`. W JavaScripcie zrobisz to za pomocą wywołania `Math.pow(3, 6)`.

Spróbujmy utworzyć własny kod obliczający potęgi. Algorytm jest prosty: wystarczy zapisać pętlę, która wielokrotnie mnoży liczbę przez nią samą, i zwrócić otrzymany wynik. Oto iteracyjny algorytm obliczania *n*-tej potęgi w Pythonie (*exponentByIteration.py*):

Python

```
def exponentByIteration(a, n):
    result = 1
    for i in range(n):
        result *= a
    return result
```

```
print(exponentByIteration(3, 6))
print(exponentByIteration(10, 3))
print(exponentByIteration(17, 10))
```

A to jego odpowiednik w JavaScripcie (*exponentByIteration.html*):

JavaScript

```
<script type="text/javascript">
function exponentByIteration(a, n) {
    let result = 1;
    for (let i = 0; i < n; i++) {
        result *= a;
    }
    return result;
}

document.write(exponentByIteration(3, 6) + "<br />");
document.write(exponentByIteration(10, 3) + "<br />");
document.write(exponentByIteration(17, 10) + "<br />");
</script>
```

Po uruchomieniu powyższych kodów otrzymasz następujące dane wyjściowe:

```
729
1000
2015993900449
```

Są to proste obliczenia, które możemy łatwo wykonać za pomocą pętli. Wadą użycia pętli jest to, że wydajność obliczeń spada wraz ze wzrostem wykładnika: obliczenie 3^{12} zajmuje dwa razy więcej czasu niż 3^6, a wyznaczenie 3^{600} trwa sto razy dłużej niż znalezienie 3^6. W następnym punkcie rozwiążemy ten problem za pomocą rekurencji.

Rekurencyjna funkcja potęgująca

Zastanów się, jak rekurencyjnie obliczyć potęgę, na przykład 3^6. Ze względu na łączność mnożenia $3{\cdot}3{\cdot}3{\cdot}3{\cdot}3{\cdot}3$ to tyle samo co $(3{\cdot}3{\cdot}3){\cdot}(3{\cdot}3{\cdot}3)$, czyli tyle co $(3{\cdot}3{\cdot}3)^2$, a ponieważ $(3{\cdot}3{\cdot}3)$ równa się 3^3, 3^6 równa się $(3^3)^2$. Jest to przykład **reguły potęgowania potęgi**: $(a^m)^n = a^{mn}$. W matematyce znana jest również **reguła mnożenia potęg o takich samych podstawach**: $a^n{\cdot}a^m = a^{n+m}$. Jej szczególny przypadek to $a^n{\cdot}a = a^{n+1}$.

Reguły te możemy wykorzystać do utworzenia funkcji `exponentByRecursion()`. Wywołanie metody `exponentByRecursion(3, 6)` będzie odpowiadało wyrażeniu `exponentByRecursion(3, 3) * exponentByRecursion(3, 3)`. Oczywiście nie musisz wykonywać obu wywołań `exponentByRecursion(3, 3)`. W zamian wystarczy po prostu zapisać zwracaną wartość w zmiennej i pomnożyć ją przez nią samą.

Metoda ta działa dla parzystych wykładników, ale co z nieparzystymi? Gdybyśmy musieli obliczyć 3^7, czyli 3·3·3·3·3·3·3, to będzie to to samo co (3·3·3·3·3·3)·3 lub (3^6)·3. Następnie, aby obliczyć 3^6, możesz wykonać to samo wywołanie rekurencyjne co poprzednio.

UWAGA *Istnieje prosta programistyczna sztuczka pozwalająca sprawdzić, czy liczba całkowita jest parzysta. Wykorzystuje się w niej operator modulo (%). Reszta z dzielenia dowolnej parzystej liczby całkowitej przez 2 to 0. Dowolna nieparzysta liczba całkowita modulo 2 daje w wyniku 1.*

Są to przypadki rekurencyjne, a jakie są przypadki bazowe? Z matematycznego punktu widzenia każda liczba do potęgi zero to 1, a każda liczba do potęgi pierwszej to po prostu ta liczba. A zatem dla dowolnego wywołania funkcji `exponentByRecursion(a, n)`, jeśli `n` ma wartość `0` lub `1`, zwracamy odpowiednio `1` lub `a`, ponieważ a^0 to zawsze 1, a a^1 to zawsze a.

Korzystając z tych wszystkich informacji, możemy zapisać kod funkcji `exponentByRecursion()`. Oto zawartość pliku *exponentByRecursion.py*, zawierającego jej implementację w Pythonie:

Python

```
def exponentByRecursion(a, n):
    if n == 1:
        # PRZYPADEK BAZOWY
        return a
    elif n % 2 == 0:
        # PRZYPADEK REKURENCYJNY (n parzyste)
        result = exponentByRecursion(a, n / 2)
        return result * result
    elif n % 2 == 1:
        # PRZYPADEK REKURENCYJNY (n nieparzyste)
        result = exponentByRecursion(a, n - 1)
        return result * a

print(exponentByRecursion(3, 6))
print(exponentByRecursion(10, 3))
print(exponentByRecursion(17, 10))
```

A to odpowiadający jej program w JavaScripcie (*exponentByRecursion.html*):

JavaScript

```
<script type="text/javascript">
function exponentByRecursion(a, n) {
    if (n === 1) {
        // PRZYPADEK BAZOWY
        return a;
    } else if (n % 2 === 0) {
        // PRZYPADEK REKURENCYJNY (n parzyste)
        result = exponentByRecursion(a, n / 2);
        return result * result;
    } else if (n % 2 === 1) {
```

```
        // PRZYPADEK REKURENCYJNY (n nieparzyste)
        result = exponentByRecursion(a, n - 1);
        return result * a;
    }
}

document.write(exponentByRecursion(3, 6) + "<br />");
document.write(exponentByRecursion(10, 3) + "<br />");
document.write(exponentByRecursion(17, 10) + "<br />");
</script>
```

Po uruchomieniu tego kodu otrzymasz dane wyjściowe identyczne jak w przypadku algorytmu iteracyjnego:

```
729
1000
2015993900449
```

Każde wywołanie rekurencyjne zmniejsza rozmiar problemu o połowę. Dzięki temu nasz algorytm rekurencyjny jest szybszy niż jego iteracyjny odpowiednik. Iteracyjne wyznaczenie 3^{1000} wymaga 1000 operacji mnożenia, podczas gdy algorytm rekurencyjny wykona w tym przypadku jedynie 23 mnożenia i dzielenia. Podczas uruchamiania kodu w Pythonie w narzędziu do profilowania iteracyjne wyznaczenie 3^{1000} 100 000 razy zajmuje 10,633 sekundy. Na wykonanie tych samych obliczeń metodą rekurencyjną potrzeba jedynie 0,406 sekundy. To ogromna poprawa!

Iteracyjne obliczanie potęgi na podstawie wniosków z algorytmu rekurencyjnego

Nasza oryginalna iteracyjna funkcja potęgująca wykorzystywała proste podejście: wykonaj pętlę tyle razy, ile wynosi wykładnik potęgi. Jednak takie podejście nie skaluje się zbyt dobrze wraz ze wzrostem wykładnika. Próba utworzenia implementacji rekurencyjnej zmusiła nas do zastanowienia się, jak podzielić ten problem na mniejsze podproblemy. Takie podejście okazało się znacznie bardziej wydajne.

Ponieważ każdy algorytm rekurencyjny ma równoważną postać iteracyjną, możemy utworzyć nową iteracyjną funkcję potęgującą opartą na regule potęgowania, z której korzysta algorytm rekurencyjny. Oto plik *exponentWithPowerRule.py*, zawierający taką funkcję zapisaną w Pythonie:

Python

```
def exponentWithPowerRule(a, n):
    # Krok 1. Ustalenie, jakie operacje należy wykonać
    opStack = []
    while n > 1:
        if n % 2 == 0:
```

```
            # n jest parzyste
            opStack.append('square')
            n = n // 2
        elif n % 2 == 1:
            # n jest nieparzyste
            n -= 1
            opStack.append('multiply')

    # Krok 2. Wykonanie operacji w odwrotnej kolejności
    result = a # Początkowo result ma wartość 'a'
    while opStack:
        op = opStack.pop()

        if op == 'multiply':
            result *= a
        elif op == 'square':
            result *= result

    return result

print(exponentWithPowerRule(3, 6))
print(exponentWithPowerRule(10, 3))
print(exponentWithPowerRule(17, 10))
```

Oto odpowiednik tego programu zapisany w JavaScripcie (znajdziesz go w pliku *exponentWithPowerRule.html*):

JavaScript

```
<script type="text/javascript">
function exponentWithPowerRule(a, n) {
    // Krok 1. Ustalenie, jakie operacje należy wykonać
    let opStack = [];
    while (n > 1) {
        if (n % 2 === 0) {
            // n jest parzyste
            opStack.push("square");
            n = Math.floor(n / 2);
        } else if (n % 2 === 1) {
            // n jest nieparzyste
            n -= 1;
            opStack.push("multiply");
        }
    }

    // Krok 2. Wykonanie operacji w odwrotnej kolejności
    let result = a; // Początkowo result ma wartość 'a'
    while (opStack.length > 0) {
        let op = opStack.pop();

        if (op === "multiply") {
            result = result * a;
        } else if (op === "square") {
```

```
            result = result * result;
        }
    }

    return result;
}

document.write(exponentWithPowerRule(3, 6) + "<br />");
document.write(exponentWithPowerRule(10, 3) + "<br />");
document.write(exponentWithPowerRule(17, 10) + "<br />");
</script>
```

Algorytm zmniejsza `n`, dzieląc je na pół (jeśli jest parzyste) lub odejmując od niego 1 (jeśli jest nieparzyste), aż do uzyskania wartości 1. W ten sposób otrzymujemy ciąg operacji podnoszenia do kwadratu lub mnożenia przez `a`, które musimy wykonać. Po zakończeniu tego kroku należy wykonać te czynności w odwrotnej kolejności. Do odwracania kolejności operacji przydaje się stos (nie mylić ze stosem wywołań), ponieważ jest to struktura danych typu *pierwsze na wejściu, ostatnie na wyjściu*. W pierwszym kroku umieszczamy na stosie `opStack` operacje podnoszenia do kwadratu lub mnożenia przez `a`. W drugim kroku wykonujemy te operacje, usuwając je ze stosu.

Na przykład wywołanie `exponentWithPowerRule(6, 5)` w celu obliczenia 6^5 powoduje przypisanie do `a` wartości `6`, a do `n` wartości `5`. Funkcja zauważa, że `n` jest nieparzyste. Oznacza to, że powinniśmy odjąć od `n` 1 (da to `4`) i odłożyć na stos `opStack` operację mnożenia przez `a`. Teraz, gdy `n` ma wartość `4` (liczba parzysta), dzielimy `n` przez `2` (otrzymujemy `2`) i odkładamy na stosie `opStack` operację podnoszenia do kwadratu. Ponieważ `n` ma wartość `2` i ponownie jest parzyste, dzielimy je przez `2` (otrzymując `1`) i odkładamy na stosie `opStack` kolejną operację podnoszenia do kwadratu. Pierwszy krok kończy się, ponieważ `n` ma wartość `1`.

Drugi krok rozpoczyna się od przypisania `a` (które ma wartość `6`) do zmiennej `result`. Zdejmujemy ze stosu `opStack` pierwszą operację, którą jest podniesienie do kwadratu, i nakazujemy programowi zmianę wartości `result` na `result * result` (inaczej `result`^{2}), czyli `36`. Następnie zdejmujemy ze stosu `opStack` kolejną operację. Jest to następna operacja podnoszenia do kwadratu, więc program zastępuje `36` ze zmiennej `result` wartością `36 * 36`, czyli `1296`. Dalej pobieramy ze stosu `opStack` ostatnią operację, którą jest mnożenie przez `a`, i mnożymy wartość `1296` znajdującą się w zmiennej `result` przez `a` (czyli 6), w wyniku czego otrzymujemy `7776`. Stos `opStack` jest teraz pusty i funkcja kończy swoje działanie. Po dokładnym sprawdzeniu obliczeń przekonasz się, że 6^5 to rzeczywiście 7776.

Zawartość stosu `opStack` w trakcie wykonywania wywołania funkcji `exponentWithPowerRule(6, 5)` pokazałem na rysunku 2.4.

Rysunek 2.4. Zawartość stosu opStack podczas wywołania funkcji exponentWithPowerRule(6, 5)

Po uruchomieniu tego kodu otrzymasz wynik identyczny jak poprzednio:

```
729
1000
2015993900449
```

Iteracyjna funkcja potęgująca wykorzystująca regułę potęgowania jest bardziej wydajna niż algorytm rekurencyjny i jednocześnie nie jest narażona na ryzyko wystąpienia błędu przepełnienia stosu. Być może nie wpadlibyśmy na ten nowy, ulepszony algorytm iteracyjny bez przeprowadzenia rozumowania rekurencyjnego.

Kiedy powinno się korzystać z rekurencji?

Nigdy *nie musisz* używać rekurencji. Rozwiązanie żadnego problemu programistycznego *nie wymaga* zastosowania rekurencji. Ten rozdział pokazał, że rekurencja nie ma żadnych magicznych zdolności robienia rzeczy, których nie może zrobić kod za pomocą iteracyjnego podejścia z użyciem pętli i stosu. W praktyce funkcja rekurencyjna może być zbyt skomplikowanym rozwiązaniem Twojego problemu.

Z drugiej strony, jak pokazuje przykład funkcji potęgującej z poprzedniego punktu, rekurencja może wzbogacić Twój tok rozumowania związany z rozwiązywanym problemem programistycznym. Podejście rekurencyjne szczególnie dobrze sprawdza się w przypadku problemów wykazujących następujące trzy cechy:

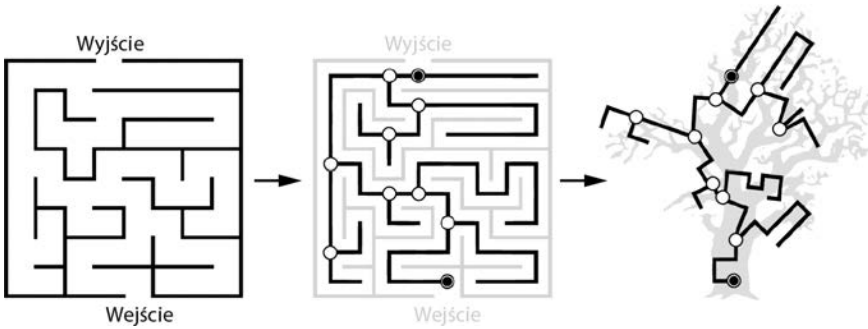
- Problem ma strukturę drzewiastą.
- Problem wymaga nawrotów.
- Głębokość rekurencji nie jest wystarczająco duża, aby wystąpił błąd przepełnienia stosu.

Drzewa charakteryzują się **samopodobieństwem** — znajdujące się w nich punkty rozgałęzień przypominają korzenie mniejszego poddrzewa. Rekurencja często wiąże się z samopodobieństwem i problemami, które można podzielić na mniejsze, podobne podproblemy. Korzeń drzewa odpowiada pierwszemu wywołaniu funkcji

rekurencyjnej, punkty rozgałęzień to przypadki rekurencyjne, a liście odpowiadają przypadkom bazowym, w których nie wykonuje się już wywołań rekurencyjnych.

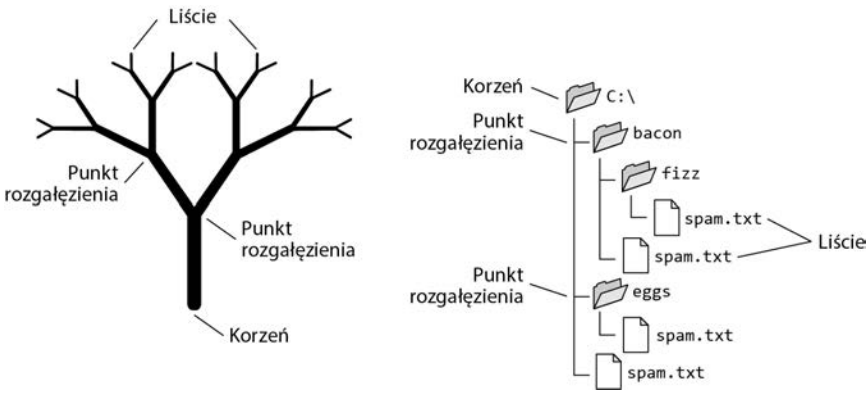
Dobrym przykładem problemu, który ma strukturę drzewiastą i wymaga nawrotów, jest labirynt. W labiryncie rozgałęzienia występują wszędzie tam, gdzie musisz wybrać jedną z wielu ścieżek, którymi chcesz podążać. Jeśli dotrzesz do ślepego zaułka, napotkałeś przypadek bazowy. Aby z niego wyjść, musisz cofnąć się do poprzedniego rozgałęzienia i wybrać inną ścieżkę.

Na rysunku 2.5 pokazałem ścieżkę w labiryncie, która została przekształcona tak, aby wyglądała jak drzewo. Pomimo wizualnych różnic pomiędzy ścieżkami w labiryncie a ścieżkami w drzewie punkty rozgałęzień w obu tych strukturach są ze sobą powiązane w ten sam sposób. Matematycy powiedzieliby, że te grafy są sobie równoważne.



Rysunek 2.5. Labirynt (po lewej) wraz z wewnętrznymi ścieżkami (w środku) przekształcony w znaną z biologii strukturę drzewa (po prawej)

U podstaw wielu problemów programistycznych leżą struktury drzewiaste. Strukturę taką ma na przykład system plików. Podkatalogi w systemie plików przypominają katalogi główne mniejszych systemów plików. Porównanie drzewa i systemu plików znajdziesz na rysunku 2.6.



Rysunek 2.6. Struktura systemu plików jest podobna do drzewa

Wyszukiwanie określonego pliku w katalogu to problem rekurencyjny. Najpierw przeszukujesz katalog, a następnie rekurencyjnie przeszukujesz znajdujące się w nim podkatalogi. Katalogi bez podkatalogów to przypadki bazowe, które powodują zatrzymanie wyszukiwania rekurencyjnego. Jeśli Twój algorytm rekurencyjny nie znajdzie pliku, którego szukasz, program cofa się do poprzedniego katalogu nadrzędnego i stamtąd kontynuuje wyszukiwanie.

Trzecia cecha ma czysto praktyczny charakter. Jeśli Twoja struktura drzewiasta zawiera tak wiele poziomów, że funkcja rekurencyjna spowodowałaby błąd przepełnienia stosu, zanim program dotrze do liści, to rekurencja nie jest odpowiednim rozwiązaniem.

Z drugiej strony wykorzystanie rekurencji jest najlepszym sposobem tworzenia kompilatorów. Projektowanie kompilatorów to obszerne zagadnienie, którego omówienie wykracza poza zakres tej książki, ale języki programowania są opisywane przez zbiór reguł gramatycznych, które pozwalają przekształcić kod źródłowy w strukturę drzewa, podobnie jak reguły gramatyczne języka polskiego pozwalają przeprowadzić rozbiór zdania w postaci drzewa. Rekurencja jest idealną techniką do zastosowania w kompilatorach.

W tej książce omówię wiele algorytmów rekurencyjnych. Sporo z nich będzie wykorzystywało strukturę drzewiastą lub nawroty, które również prowadzą do rekurencji.

Tworzenie algorytmów rekurencyjnych

Mam nadzieję, że ten rozdział dał Ci pewne wyobrażenie o tym, jak funkcje rekurencyjne wypadają w porównaniu z algorytmami iteracyjnymi, które prawdopodobnie znasz o wiele lepiej. W dalszej części książki zagłębimy się w szczegóły różnych algorytmów rekurencyjnych. Być może zastanawiasz się, jak utworzyć swoją własną funkcję rekurencyjną.

Pierwszym krokiem jest zawsze zidentyfikowanie przypadku rekurencyjnego i przypadku bazowego. Możesz zastosować podejście odgórne polegające na podziale problemu na podproblemy podobne do problemu pierwotnego, ale od niego mniejsze. Podział ten odpowiada *przypadkowi rekurencyjnemu*. Następnie zastanów się, kiedy podproblemy są wystarczająco małe, aby rozwiązać je w sposób trywialny. Taki podproblem to *przypadek bazowy*. Twoja funkcja rekurencyjna może mieć więcej niż jeden przypadek rekurencyjny lub więcej niż jeden przypadek bazowy, ale wszystkie funkcje rekurencyjne zawsze muszą mieć co najmniej jeden przypadek rekurencyjny i co najmniej jeden przypadek bazowy.

Przykładem takiego algorytmu jest rekurencyjny algorytm wyznaczania wyrazów ciągu Fibonacciego. Liczba Fibonacciego jest sumą dwóch poprzednich liczb Fibonacciego. Problem znalezienia *n*-tej liczby Fibonacciego można rozbić na dwa podproblemy znalezienia dwóch mniejszych liczb Fibonacciego. Wiemy, że pierwsze dwie liczby Fibonacciego to 1, a zatem mamy przypadki bazowe odpowiadające wystarczająco małym podproblemom.

Czasami pomocne jest przyjęcie podejścia oddolnego i rozważenie przypadku bazowego, a następnie zaobserwowanie, jak konstruowane i rozwiązywane są na jego podstawie coraz to większe problemy. Przykładem jest rekurencyjny algorytm wyznaczania silni. 1! to 1. Jest to to przypadek bazowy. Następna silnia to 2!, którą wyznacza się poprzez pomnożenie 1! przez 2. Kolejna silnia, czyli 3!, jest wyznaczana przez pomnożenie 2! i 3 i tak dalej. Na podstawie tego ogólnego wzorca możemy wyznaczyć przypadek rekurencyjny tworzonego algorytmu.

Podsumowanie

W tym rozdziale omówiłem obliczanie silni i wyznaczanie ciągu Fibonacciego — dwa klasyczne problemy programowania rekurencyjnego. Opisałem zarówno iteracyjne, jak i rekurencyjne rozwiązania tych problemów. Pomimo tego, że są to klasyczne przykłady rekurencji, związane z nimi algorytmy rekurencyjne mają poważne wady. Rekurencyjna implementacja silni może powodować przepełnienie stosu, podczas gdy rekurencyjna funkcja Fibonacciego wykonuje tak wiele zbędnych obliczeń, że jest zbyt wolna, aby mieć jakieś praktyczne zastosowanie.

W tym rozdziale wyjaśniłem też, jak przekształcić algorytm rekurencyjny w iteracyjny i odwrotnie. Algorytmy iteracyjne wykorzystują pętlę, a każdy algorytm rekurencyjny może być wykonany iteracyjnie przy użyciu pętli i stosu. Rekurencja jest często zbyt skomplikowanym rozwiązaniem, ale technika ta sprawdza się szczególnie dobrze w problemach, które mają strukturę drzewiastą i wykorzystują nawroty.

Pisanie funkcji rekurencyjnych to umiejętność, której rozwój wymaga praktyki i doświadczenia. W dalszej części tej książki omówię kilka dobrze znanych przykładów rekurencji oraz przedstawię ich mocne i słabe strony.

Materiały dodatkowe

Więcej informacji na temat różnic pomiędzy podejściami iteracyjnym i rekurencyjnym znajdziesz w filmie *Programming Loops vs. Recursion* z kanału Computerphile w serwisie YouTube (*https://youtu.be/HXNhEYqFo0o*). Jeśli chcesz porównywać wydajność funkcji iteracyjnych i rekurencyjnych, musisz nauczyć się korzystać z profilera. Profilery Pythona są opisane w rozdziale 13. mojej książki *Programowanie w Pythonie dla średnio zaawansowanych. Najlepsze praktyki tworzenia czystego kodu* (Helion, Gliwice 2021). Rozdział ten (w języku angielskim) znajdziesz również na stronie *https://inventwithpython.com/beyond/chapter13.html*.

Profilery omówiono też w oficjalnej dokumentacji Pythona (*https://docs.python.org/3/library/profile.html*). Dostępny w Firefoksie profiler języka JavaScript jest opisany na stronie internetowej Mozilli (*https://profiler.firefox.com/docs/#/*). Inne przeglądarki zawierają profilery podobne do tego z Firefoksa.

Pytania praktyczne

Sprawdź swoją wiedzę, odpowiadając na poniższe pytania:

1. Co to jest 4! (cztery silnia)?
2. Jak wykorzystać silnię z $(n - 1)$ w obliczeniach silni z n?
3. Jaka jest największa słabość rekurencyjnej implementacji silni?
4. Wymień pierwsze pięć wyrazów ciągu Fibonacciego.
5. Jakie dwie liczby należy do siebie dodać, aby otrzymać n-tą liczbę Fibonacciego?
6. Jaka jest największa słabość rekurencyjnej implementacji algorytmu wyznaczania ciągu Fibonacciego?
7. Z czego zawsze korzysta algorytm iteracyjny?
8. Czy zawsze można przekształcić algorytm iteracyjny w rekurencyjny?
9. Czy zawsze można przekształcić algorytm rekurencyjny w iteracyjny?
10. Jakie dwa elementy są wymagane do wykonania algorytmu rekurencyjnego w sposób iteracyjny?
11. Jakie trzy cechy charakteryzują problemy, które warto rozwiązać rekurencyjnie?
12. Kiedy należy rozwiązać problem programistyczny za pomocą rekurencji?

Zadania

W ramach ćwiczeń utwórz funkcje rozwiązujące poniższe zadania:

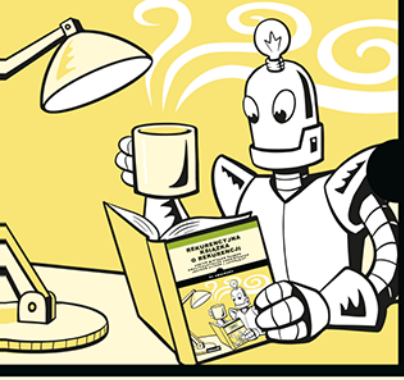
1. Oblicz w sposób iteracyjny sumę liczb całkowitych od 1 do n. Algorytm będzie podobny do tego z funkcji `factorial()` z tą różnicą, że w tym przypadku zamiast mnożenia należy wykonać dodawanie. Na przykład `sumSeries(1)` powinno zwracać `1`, `sumSeries(2)` powinno zwracać `3` (czyli `1 + 2`), `sumSeries(3)` powinno zwracać `6` (czyli `1 + 2 + 3`) i tak dalej. Utworzona przez Ciebie funkcja powinna korzystać z pętli, a nie z rekurencji. Po więcej wskazówek zajrzyj do pliku *factorialByIteration.py* z tego rozdziału.
2. Przekształć funkcję `sumSeries()` w funkcję rekurencyjną, która zamiast z pętli korzysta z wywołań rekurencyjnych. Po więcej wskazówek zajrzyj do pliku *factorialByRecursion.py* z tego rozdziału.
3. Utwórz funkcję `sumPowersOf2()`, która oblicza iteracyjnie sumę pierwszych n potęg liczby 2. Potęgi dwójki to 2, 4, 8, 16, 32 itd. W Pythonie wartości te można obliczyć jako `2 ** 1`, `2 ** 2`, `2 ** 3`, `2 ** 4`, `2** 5` itd.

W JavaScripcie możesz je uzyskać z wywołań `Math.pow(2, 1)`, `Math.pow(2, 2)` itd. Wywołanie `sumPowersOf2(1)` powinno zwracać `2`, `sumPowersOf2(2)` powinno zwracać `6` (czyli 2 + 4), a `sumPowersOf2(3)` powinno zwracać `14` (czyli 2 + 4 + 8) itd.

4. Przekształć funkcję `sumPowersOf2()` w funkcję rekurencyjną. Funkcja ta powinna korzystać z wywołań funkcji rekurencyjnych zamiast z pętli.

Notatki


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



ZANIM ZROZUMIESZ REKURENCJĘ,
MUSISZ NAJPIERW...
ZROZUMIEĆ REKURENCJĘ!

PRZYGOTUJ SWÓJ MÓZG NA NIEZŁĄ GIMNASTYKĘ!

— DAVID BEAZLEY, LEGENDA PYTHONA, DWUKROTNY LAUREAT IEEE GORDON BELL PRIZE

Rekurencja jest świetna — co więcej, dla Ciebie może oznaczać udaną rozmowę kwalifikacyjną! To metoda pomocna w rozwiązywaniu trudnych zagadnień: sprowadza złożone problemy do znacznie łatwiejszych. Myślenie rekurencyjne przydaje się często podczas projektowania oprogramowania, nawet jeśli nie stosuje się w nim wprost rekurencji. Wielu twórców oprogramowania jej unika, uważa ją bowiem za trudną i niezrozumiałą. Przekonaj się, że jest inaczej!

Dzięki tej książce zrozumiesz, że w rekurencji nie kryje się żadna magia. Dowiesz się, na czym polega jej działanie i kiedy warto zastosować algorytm rekursywny, a kiedy lepiej tego nie robić. Poznasz szereg klasycznych i mniej znanych algorytmów rekurencyjnych. Pracę z zawartym tu materiałem ułatwią Ci liczne przykłady programów napisanych w Pythonie i JavaScripcie, pokazujące, jak rozwiązywać przeróżne problemy związane z przechodzeniem przez drzewa, kombinatoryką i innymi trudnymi zagadnieniami. Nauczysz się także skutecznie poprawiać wydajność kodu i algorytmów rekurencyjnych.

Sprawdź i zrozum:

- czym jest rekurencja i jak działają klasyczne algorytmy rekurencyjne języka C
- w jaki sposób funkcje rekurencyjne wykorzystują stos wywołań
- jak rekurencja ogonowa upraszcza pisanie funkcji rekurencyjnych
- dlaczego rekurencja ułatwia rozwiązywanie niestandardowych problemów
- pisanie poprawnych procedur obsługi przerwań

Al Sweigart jest programistą, członkiem Python Software Foundation i autorem książek informatycznych. Opracował kilka popularnych modułów dla Pythona. Tworzone przez siebie oprogramowanie często udostępnia na zasadach open source.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-8322-653-8
9 788383 226538

Cena: 79,00 zł

no starch press