MATEMATYKA I INFORMATYKA NA UCZELNIACH TECHNICZNYCH

MINUT 2021 (3), s. 87–97
ISSN 2719-3063

Robert TUTAJEWICZ[1]

[1]Katedra Informatyki Stosowanej, Politechnika Śląska, ul. Akademicka 16, 44-100 Gliwice

Wprowadzenie do algorytmów rekurencyjnych

Streszczenie. Artykuł prezentuje, czym jest rekurencja, jakie są jej mocne i słabe strony. Zostało w nim także zaprezentowanych i omówionych kilka prostych algorytmów rekurencyjnych. W artykule przedstawiono rekurencyjne i iteracyjne wersje algorytmów potęgowania, obliczania silni, obliczania wyrazów ciągu Fibonacciego, a także rekurencyjne wersje rozwiązania problemów wież Hanoi i reprezentacji liczby naturalnej w postaci sumy naturalnych składników.

Słowa kluczowe: algorytm, rekurencja, złożoność obliczeniowa, silnia, ciąg Fibonacciego, wieża Hanoi.

1. Wstęp

Rekurencja jest jednym z podstawowych pojęć używanych w informatyce. Świadomie zastosowana staje się bardzo mocnym narzędziem programisty. Jeśli jednak używa się jej nieodpowiednio, może stać się przyczyną wielu kłopotów. Dlatego też ważne jest, aby zrozumieć zasadę przetwarzania rekurencyjnego na samym początku kształcenia inżyniera informatyka.

Niniejszy artykuł jest próbą przystępnego wyjaśnienia pojęcia rekurencji. Na kilku prostych przykładach zostaną pokazane zalety podejścia rekurencyjnego, a także zasygnalizowane słabe strony rozwiązań rekurencyjnych. Prezentowane algorytmy zapisywane są w postaci fragmentów kodu w języku C. Zakłada się, że czytelnik w co najmniej podstawowym zakresie zna ten język lub inny język programowania wywodzący się z języka C.

Warto zauważyć, że rekurencja nie została wymyślona przez człowieka, ale występuje w przyrodzie niemal od zawsze. Rekurencyjne wzory można odnaleźć u wielu roślin i zwierząt. Przykładem mogą być chociażby zwinięte planspiralnie (w jednej płaszczyźnie) muszle morskich głowonogów z rzędu łodzików (*Nautilida*), czy cechujące się samopodobieństwem liście paproci. Nawet w świecie nieożywionym można odnaleźć przykłady rekurencji. Wystarczy na przykład przyjrzeć się bliżej płatkom padającego zimą śniegu, by dopatrzeć się powtarzających się kształtów.

Autor korespondencyjny: R. Tutajewicz (robert.tutajewicz@polsl.pl).
Data wpłynięcia: 17.02.2021.

2. Proste przykłady rekurencji

Istota rekurencji polega na wykorzystaniu w rozwiązaniu problemu założenia, że potrafimy ten sam problem rozwiązać tylko dla nieco mniejszej skali (np. dla mniejszej wartości danej wejściowej). Jeżeli z tego założenia potrafimy wywieść rozwiązanie dla większej skali, otrzymujemy rozwiązanie rekurencyjne. Przykładem takiego sposobu myślenia może być obliczanie n-tej potęgi liczby x, gdzie $n \in \mathbb{N}$. Aby policzyć x^n, należy najpierw wyznaczyć wartość x^{n-1} i następnie tę wartość pomnożyć przez x. Ponieważ jednak nie znamy wartości x^{n-1}, do jej policzenia możemy zastosować tę samą metodę. Taki sposób postępowania powtarzamy aż do momentu, gdy następuje potrzeba wyznaczenia wartości x^1. Oczywistym jest że $x^1 = x$ i z tej własności korzystamy[1].

Przykładowo załóżmy, że chcemy policzyć wartość 3^5. Ponieważ nie wiemy, ile to jest, najpierw musimy wyliczyć 3^4 a następnie uzyskaną wartość pomnożyć przez 3. Do policzenia 3^4 musimy najpierw obliczyć 3^3 i pomnożyć obliczoną wartość przez 3. Aby wyznaczyć 3^3, trzeba najpierw ustalić, ile to jest 3^2, a to z kolei wymaga przemnożenia $3^1 \cdot 3$. Wreszcie określenie wartości 3^1 jest zadaniem trywialnym. Dla wszystkich jest oczywiste, że jest to 3. W tym przypadku nie musimy już odwoływać się do niższej potęgi trójki, ale za wynik wprost przyjmujemy wartość 3. Taka sytuacja jest czymś naturalnym w przetwarzaniu rekurencyjnym i nazywamy ją *przypadkiem bazowym* lub przypadkiem szczególnym. Obok przypadku bazowego występuje *przypadek ogólny*. W omawianym przykładzie odnosi się on do tych miejsc, gdzie posługujemy się mnożeniem niższej potęgi przez 3.

Formalnie obliczanie wartości naturalnej potęgi liczby x można zapisać za pomocą równania

$$f(x) = \begin{cases} x & \text{dla } n = 1, \\ x^{n-1} \cdot x & \text{dla } n > 1. \end{cases} \quad (1)$$

Równanie to może być podstawą do zdefiniowania odpowiedniej funkcji w języku C.

```
1 double recpower(double x, int n)
2 {
3     if ( n > 1 )
4         return recpower(x, n - 1) * x;
5     else
6         return x;
7 }
```

Listing 1. Rekurencyjna wersja funkcji obliczającej potęgę

Za pomocą funkcji o nazwie `recpower` można obliczyć naturalną potęgę liczby x w sposób rekurencyjny (o ile tylko jako parametr `n` zostanie podstawiona liczba naturalna większa lub równa 1). Można w niej wyróżnić obydwa elementy typowe dla funkcji rekurencyjnych. Linia 4 kodu to przypadek ogólny, gdzie do wyliczenia większej potęgi liczby korzystamy ze znajomości potęgi mniejszej (czyli wywołujemy ponownie funkcję `recpower`). W linii szóstej występuje zaś przypadek bazowy, dla którego rozwiązanie jest oczywiste i nie jest wymagane wywoływanie funkcji rekurencyjnej.

Jednym z błędów popełnianych przez początkujących programistów jest zbytnie koncentrowanie się na przypadku ogólnym i w konsekwencji pomijanie przypadku bazowego. Listing 2 przedstawia taką sytuację.

[1] Ponieważ zero w matematyce jest czasem traktowane jako liczba naturalna, a czasem nie, w opracowaniu niniejszym przyjęto, że liczbami naturalnymi są wyłącznie liczby całkowite większe od zera.

```
1 double recpower(double x, int n)
2 {
3     return recpower(x, n - 1) * x;
4 }
```

Listing 2. Pominięty przypadek bazowy

W stosunku do wersji poprawnej (listing 1) usunięto tu sprawdzanie warunku (linia 3 na listingu 1), w efekcie czego obliczenia będą trwały niemal w nieskończoność[2]. Przeanalizujmy dokładniej, jak w takim przypadku będzie obliczana wartość funkcji `recpower(3.0, 2)`. Zgodnie z kodem funkcji należy policzyć tę wartość jako `recpower(3.0, 1) * 3.0`. Żeby wyznaczyć wartość `recpower(3.0, 1)` trzeba wpierw policzyć, ile to jest `recpower(3.0, 0)`, co z kolei wymaga wyznaczenia wartości `recpower(3.0, -1)` i tak dalej. W żadnym momencie przetwarzania nie wystepuje sytuacja przerywająca ten nieskończony łańcuch wywołań. Obliczenia będą wykonywane aż do wyczerpania zasobów komputera.

Spostrzegawczy Czytelnik zauważy zapewne, że potęgę można obliczyć także bez użycia rekurencji. Przecież

$$x^n = \prod_{i=1}^{n} x, \tag{2}$$

a taką operację obliczania iloczynu można bardzo łatwo zrealizować za pomocą pętli `for`, co zaprezentowano na listingu 3. Rozwiązanie takie określa się mianem metody iteracyjnej.

```
1 double itpower(double x, int n)
2 {
3     double result = 1.0;
4     for (int i = 1; i <= n; i++)
5         result = result * x;
6     return result;
7 }
```

Listing 3. Iteracyjna wersja funkcji obliczającej potęgę

Najczęściej chyba podawanym przykładem algorytmu rekurencyjnego jest algorytm obliczania silni. Silnię rekurencyjnie można zdefiniować następująco:

$$n! = \begin{cases} 1 & \text{dla } n = 0, \\ n \cdot (n-1)! & \text{dla } n \geqslant 1. \end{cases} \tag{3}$$

Bazując wprost na tej definicji można zaproponować rekurencyjną funkcję obliczającą silnię:

```
1 int factorial(int n)
2 {
3     if (n >= 1)
4         return n * factorial(n - 1);
5     else
6         return 1;
7 }
```

Listing 4. Rekurencyjna wersja funkcji obliczającej silnię

Warto zauważyć, że funkcja ta jest bardzo podobna do funkcji `recpower` liczącej potęgę (zob. listing 1). Należy zatem oczekiwać, że także w tym przypadku możliwe jest znalezienie rozwiązania znajdującego

[2]Próba wykonania takiego kodu zakończy się błędem spowodowanym przepełnieniem stosu, wynikającym z braku możliwości utworzenia nieskończonej liczby zmiennych w pamięci komputera.

silnię metodą iteracyjną. Tak rzeczywiście jest. Dla liczb całkowitych dodatnich silnię można obliczyć ze wzoru:

$$n! = \prod_{i=1}^{n} i, \qquad (4)$$

a wzór ten przekłada się wprost na funkcję liczącą silnię w sposób iteracyjny:

```
int factorial2(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
        result = result * i;
}
```

Listing 5. Iteracyjna wersja funkcji obliczającej silnię

Bez względu na sposób zapisu obydwie funkcje (rekurencyjna i iteracyjna) do wyznaczenia wartości silni używają dokładnie tych samych działań. Jest to wielokrotne mnożenie kolejnych liczb całkowitych począwszy od 1. Przyjrzyjmy się, jak wyznaczana będzie wartość rekurencyjnej funkcji `factorial` dla $n = 4$.

```
factorial(4) =
4 * factorial(3) =
4 * 3 * factorial(2) =
4 * 3 * 2 * factorial(1) =
4 * 3 * 2 * 1 * factorial(0)=
4 * 3 * 2 * 1 * 1.
```

Porównajmy to ze sposobem, w jaki zmienia się wartość zmiennej `result` podczas obliczania silni iteracyjnie. Zmiennej tej nadawana jest wartość początkowa równa 1. Następnie w pętli zmienna jest mnożona przez kolejne wartości począwszy od 1 a skończywszy na n. Końcowa wartość zmiennej `result` zwracana jako wynik funkcji jest zatem wynikiem następujących obliczeń

```
result = 1 * 1 * 2 * 3 * 4.
```

Znaczy to, że w obydwu przypadkach wynik jest liczony dokładnie tak samo. Tak w wersji rekurencyjnej, jak i iteracyjnej, znalezienie wyniku wiąże się z wykonaniem tej samej liczby mnożeń, na tych samych argumentach. Jeżeli uznamy, że mnożenie w tym problemie jest operacją dominującą i pominiemy pozostałe mniej istotne operacje, możemy stwierdzić, że obydwie funkcje liczące silnię są równie dobre. W praktyce jednak zaniedbanie wpływu pozostałych operacji na czas trwania obliczeń nie jest właściwe. Wywołanie każdej funkcji (także rekurencyjnej) obciążone jest dużym narzutem czasowym i dlatego, gdy mamy do dyspozycji dwie wersje funkcji rekurencyjną i iteracyjną działające tak samo, zawsze należy wybierać wersję iteracyjną. Jeżeli zauważy się, że każdy program (także ten zawierający funkcje rekurencyjne) jest kompilowany do kodu binarnego wykonywanego przez procesor sekwencyjnie, można dojść do wniosku, że zawsze można podać wersję sekwencyjną dla danego algorytmu rekurencyjnego. Jeśli nawet coś takiego jest prawdą, to znalezienie sekwencyjnej wersji rekurencyjnych algorytmów nie zawsze jest tak proste, jak miało to miejsce w przypadku potęgowania, czy obliczania silni.

3. Ciąg Fibonacciego

W 1202 roku matematyk Leonardo z Pizy zwany Fibonaccim napisał rozprawę *Liber Abaci*, w której znalazło się między innymi zadanie o królikach o następującej treści: *Ile będzie po roku par królików, które urodzą się jako potomstwo jednej pary, jeśli każda para wydaje na świat co miesiąc nową parę, zdolną z kolei po miesiącu do rozmnażania, i jeśli żadna para w tym czasie nie ginie?*

Aby rozwiązać zadanie przyjmijmy, że pierwsza para narodziła się w styczniu. Po miesiącu (czyli w lutym) pierwsza para królików osiąga dojrzałość i w marcu rodzi się kolejna para, która osiągnie dojrzałość po miesiącu (w kwietniu), a po dwóch (w maju) wyda na świat kolejną parę. Pierwsza para począwszy od marca co miesiąc będzie generować kolejną parę młodych. Także każda kolejna para po dwóch miesiącach od urodzenia będzie co miesiąc rodzić swoje młode i tak dalej. W efekcie otrzymujemy następujące liczby par królików w kolejnych miesiącach (zob. tabela 1). W każdym kolejnym miesiącu liczba par królików wzrasta w stosunku do miesiąca poprzedniego dokładnie o liczbę par, jakie były o 2 miesiące wcześniej (jako że te właśnie pary mają wtedy młode).

Tabela 1. Liczby par królików w kolejnych miesiącach

Miesiąc	Liczba par królików
Styczeń	1
Luty	1
Marzec	2
Kwiecień	3
Maj	5
Czerwiec	8
Lipiec	13
Sierpień	21
Wrzesień	34
Październik	55
Listopad	89
Grudzień	144

Uzyskany w ten sposób ciąg liczbowy, którego elementami są liczby par królików w kolejnych miesiącach, nazywany jest ciągiem Fibonacciego. Rekurencyjny wzór na kolejne wyrazy tego ciągu jest następujący:

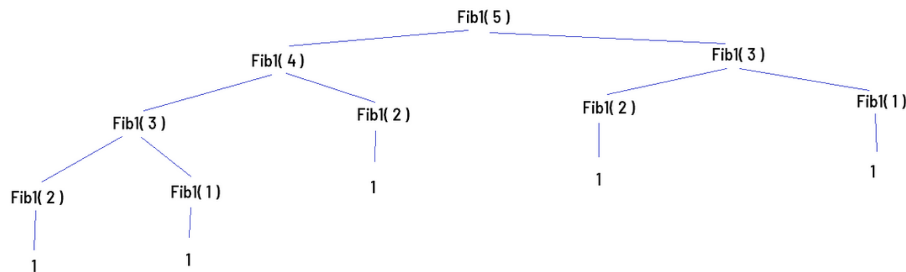
$$fib(n) = \begin{cases} 1 & \text{dla } n < 3, \\ fib(n-2) + fib(n-1) & \text{dla } n \geqslant 3. \end{cases} \tag{5}$$

Wzór 5 można zapisać w postaci funkcji w języku C:

```c
int fib(int n)
{
    if (n < 3)
        return 1;
    else
        return fib(n - 2) + fib(n - 1);
}
```

Listing 6. Rekurencyjna wersja funkcji obliczającej n-ty wyraz ciągu Fibonacciego

Przedstawiona na listingu 6 funkcja oblicza wartość n-tego wyrazu ciągu Fibonacciego w sposób niewydajny. W przypadku, gdy n przekracza 3, funkcja ta, dla niektórych wartości parametru, oblicza tę samą wartość wielokrotnie. Na rysunku 1 przedstawiono drzewo wywołań funkcji `fib` dla parametru n równego 5.



Rysunek 1. Drzewo wywołań podczas obliczania wartości funkcji Fib(5)

Wyznaczenie wartości `fib(5)` wymaga zsumowania `fib(4)` i `fib(3)`. Z kolei aby znaleźć `fib(4)` trzeba wyliczyć `fib(3)` i `fib(2)`. Wartość `fib(3)` jest zatem liczona dwukrotnie. Wartość `fib(2)`, jak wynika z drzewa wywołań, jest wyznaczana nawet trzykrotnie. Co gorsza im większa wartość parametru n, tym więcej pojawia się wielokrotnie obliczanych wartości funkcji. To niewydajne działanie (bez zapamiętywania wcześniej już policzonych wartości) jest jednym z mankamentów rozwiązań rekurencyjnych.

Opisywana ułomność nie występuje przy iteracyjnym wyznaczaniu elementów ciągu Fibonacciego. Jednak napisanie funkcji iteracyjnej nie jest już tak trywialne, jak było to w przypadku potęgowania, czy liczenia silni. We wcześniejszych przypadkach wystarczyło użyć jednej zmiennej i przechowywać w niej poprzednią wartość częściowego wyniku. Tym razem potrzeba będzie większej liczby zmiennych. Listing 7 prezentuje pierwsze przybliżenie funkcji iteracyjnej obliczającej wyrazy ciągu Fibonacciego.

```
1  int fib1(int n)
2  {
3      int x1 = 1;  // poprzedni wyraz ciągu Fibonacciego
4      int x2 = 1;  // aktualny wyraz ciągu Fibonacciego
5      for (int i = 3; i <= n; i++) {
6          int tmp = x1;
7          x1 = x2;
8          x2 = tmp + x2;
9      }
10     return x2;
11 }
```

Listing 7. Iteracyjna wersja funkcji obliczającej n-ty wyraz ciągu Fibonacciego

Funkcja `fib1` wymaga dwu zmiennych przechowujących wartości między kolejnymi przebiegami pętli. Zmienna `x2` na koniec każdego przebiegu pętli przechowuje wartość aktualnego wyrazu ciągu, zaś `x1` wartość wyrazu poprzedniego. W kolejnym przejściu pętli wcześniejszy wyraz aktualny staje się wyrazem poprzednim a poprzedni wyrazem poprzedzającym poprzedni. Aktualny wyraz w kolejnym przebiegu ma być wynikiem dodania dwóch elementów poprzednich. Dodawanie to jest wykonywane w linii 8 załączonego na listingu 7 kodu. Ponieważ linię wcześniej zmienna `x1` przyjmuje już nową wartość `x1`, w dodawaniu używa się zmiennej tymczasowej `tmp`, w której wcześniej zapamiętano poprzednią wartość `x1`. W praktyce zatem potrzeba trzech zmiennych.

Kod funkcji `fib1` można jednak nieco zmodyfikować pozbywając się zmiennej tymczasowej `temp`. Tę zmodyfikowaną wersję przedstawiono na listingu 8 i właśnie ta wersja będzie dalej analizowana.

```
int fib2(int n)
{
    int x1 = 1;
    int x2 = 1;
    for (int i = 3; i <= n; i++) {
        x2 = x1 + x2;
        x1 = x2 - x1;
    }
    return x2;
}
```

Listing 8. Zmodyfikowana iteracyjna wersja funkcji obliczającej n-ty wyraz ciągu Fibonacciego

W ciele pętli w funkcji `fib2` występują dwie instrukcje. Pierwsza (linia 6) to wyznaczenie nowej wartości aktualnej `x2` jako sumy dwu poprzednich wartości (czyli starej wartości `x2` i starej wartości `x1`). W drugiej instrukcji (linia 7) ustalana jest nowa wartość zmiennej `x1`. Zmienna ta powinna teraz przechowywać wartość `x2` z poprzedniego kroku. Niestety wartość ta nie jest już bezpośrednio dostępna, została zniszczona w wyniku poprzedniej instrukcji. Dlatego też nowa wartość `x1` jest wyliczana jako różnica nowej wartości `x2` i starej wartości `x1`.

Ponieważ mamy dwie różne wersje funkcji obliczających wartości wyrazów ciągu Fibonacciego: rekurencyjną `fib` (listing 6) i iteracyjną `fib2` (listing 8), należy je ze sobą porównać aby wybrać lepszą. Kryterium porównania będzie liczba operacji tzw. addytywnych, czyli operacji dodawania i odejmowania, wykonywanych w celu wyznaczenia wartości danego wyrazu ciągu. Z drzewa wywołań (rysunek 1) można wywnioskować, że w wersji rekurencyjnej wyznaczenie wartości n-tego wyrazu ciągu Fibonacciego sprowadza się do zsumowania jedynek, w liczbie równej wartości danego wyrazu ciągu. Do dodania do siebie $fib(n)$ jedynek potrzeba $fib(n) - 1$ dodawań. Zatem liczba dodawań potrzebnych do obliczenia n-tego wyrazu ciągu Fibonacciego w wersji rekurencyjnej jest zawsze o 1 mniejsza od wartości tego wyrazu.

W wersji iteracyjnej (funkcja `fib2`, listing 8) w każdym przebiegu pętli wykonywane jest jedno dodawanie (linia 6) i jedno odejmowanie (linia 7), tym samym łączna liczba operacji addytywnych potrzebnych do wyznaczenia n-tego wyrazu ciągu Fibonacciego w tym przypadku będzie równa $2 \cdot (n - 2)$ (pętla wykonywana jest $n - 2$ razy).

Tabela 2. Liczby operacji addytywnych potrzebnych do wyznaczenia dziesięciu pierwszych wyrazów ciągu Fibonacciego

Algorytm	1	2	3	4	5	6	7	8	9	10
rekurencyjny	0	0	1	2	4	7	12	20	33	54
iteracyjny	0	0	2	4	6	8	10	12	14	16

Tabela 2 zawiera porównanie liczby operacji addytywnych potrzebnych do wyznaczenia dziesięciu pierwszych wyrazów ciągu Fibonacciego dla obydwu rozpatrywanych wersji. Dla początkowych wartości n rozwiązanie rekurencyjne wymaga mniejszej liczby operacji niż iteracyjne. Jednakże wraz ze wzrostem n obserwuje się coraz to większą przewagę rozwiązania iteracyjnego. Bardzo często bywa tak, że dla niewielkiego rozmiaru zadania zarówno metoda iteracyjna jak i rekurencyjna sprawdza się całkiem dobrze. Ale dla danych większych rozmiarów czas obliczeń metodą rekurencyjną staje się nieakceptowalny.

Dla pozostałych dwóch przykładów zostanie przedstawione już tylko rozwiązanie rekurencyjne. Przykłady te pokazują, w jakich sytuacjach użycie rekurencji jest uzasadnione.

4. Reprezentacja liczby w postaci sumy naturalnych składników

Należy podać algorytm, obliczający na ile sposobów można przedstawić liczbę naturalną w postaci sumy naturalnych składników, przy czym rozwiązania różniące się tylko położeniem składników uważamy za identyczne. Przykładowo dla liczby 5 będą to następujące sumy: 5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1 i wreszcie 1+1+1+1+1, razem 7 sposobów.

Znalezienie algorytmu rozwiązującego to zadanie nie jest takie oczywiste. Na początek pozornie utrudnijmy sobie zadanie i postawmy je następująco: na ile sposobów można przedstawić liczbę m za pomocą składników nie większych niż n. Zacznijmy od analizy zadania i znalezienia reguł rządzących tą dziedziną. Jeśli $m = n$, to mamy przypadek postawionego zadania. Oznaczmy $f(m,n)$ funkcję określającą na ile sposobów można przedstawić liczbę m używając do tego składników nie większych od n. Jest oczywistym, że nigdy nie zostaną użyte składniki większe od samego m, czyli:

$$f(m,n) = f(m,m) \text{ gdy } m < n. \tag{6}$$

Występuje tylko jedna reprezentacja liczby m używająca jej samej, pozostałe używają tylko liczb mniejszych od m, co pozwala zapisać kolejną regułę

$$f(m,n) = 1 + f(m,m-1) \text{ gdy } m \leqslant n. \tag{7}$$

Wreszcie gdy $m > n$ prawdziwe jest, że do przedstawienia liczby m można używać składnika n i takich reprezentacji m będzie tyle, na ile sposobów da się przedstawić resztę $m - n$ przy użyciu składników o wartości co najwyżej n oraz dodatkowo należy uwzględnić te reprezentacje liczby m, które używają wyłącznie składników mniejszych od n (a tych jest $f(m,n-1)$), a zatem można zapisać regułę

$$f(m,n) = f(m-n,n) + f(m,n-1). \tag{8}$$

Dodajmy jeszcze, że liczbę 1 da się przedstawić wyłącznie na jeden sposób (1) oraz, że każdą liczbę naturalną da się przedstawić w postaci samych jedynek na jeden sposób. Z podanych reguł bezpośrednio można wyprowadzić funkcję obliczającą wartość $f(m,n)$ dla dowolnych wartości parametrów m i n. Funkcję tę prezentuje listing 9. Czyż nie jest to rozwiązanie proste i eleganckie?

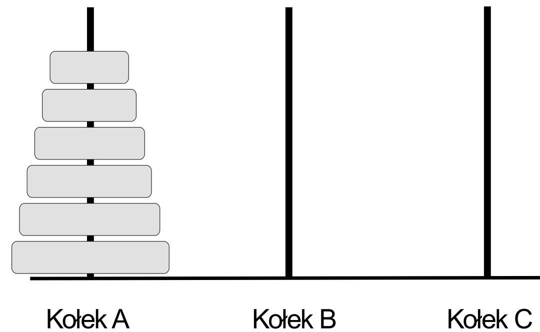
```
1 int f( int m, int n )
2 {
3     if(n==1 || m==1)
4         return 1;
5     else if( m > n )
6         return f( m - n, n ) + f( m, n - 1 );
7     else
8         return 1 + f( m, m - 1 );
9 }
```

Listing 9. Funkcja obliczająca, na ile sposobów da się przedstawić liczbę m w postaci sumy składników nie większych niż n

5. Wieża Hanoi

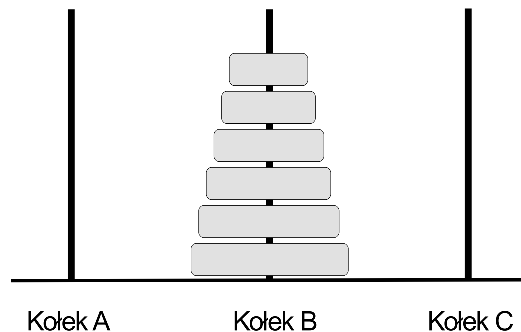
W 1883 roku francuski matematyk, zajmujący się między innymi ciągami Fibonacciego, Éduard Lucas upowszechnił łamigłówkę nazwaną wieżą Hanoi. Jej treść jest następująca: *Dane są trzy kołki (oznaczone*

jako A, B i C) oraz n krążków, każdy o innej średnicy. Na starcie wszystkie krążki są nałożone na pierwszy kołek A w kolejności malejącej od największego na dole do najmniejszego na górze. Zadanie polega na przeniesieniu wszystkich krążków z kołka A na kołek B. W trakcie wykonywania zadania należy przestrzegać kilku prostych reguł. Kołek C należy traktować jako kołek pomocniczy. W danym momencie można przenosić tylko jeden krążek. Krążki na kołkach można umieszczać tylko w kolejności malejącej (każdy krążek położony na innym krążku musi być mniejszy od tego, który jest niżej).



Rysunek 2. Stan początkowy dla zadania o wieży Hanoi dla 6 krążków

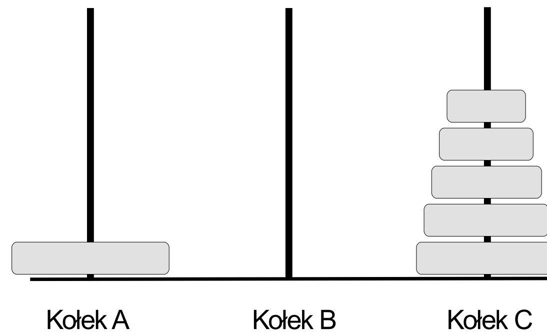
Na rysunku 2 przedstawiono początkowe ułożenie krążków dla n równego 6, a na rysunku 3 stan, do jakiego należy doprowadzić na koniec. Jak widać, na początku wszystkie krążki znajdują się na kołku A, na koniec wszystkie krążki powinny znaleźć się na kołku B.



Rysunek 3. Stan końcowy dla zadania o wieży Hanoi dla 6 krążków

Rekurencyjne podejście do problemu prowadzi do bardzo prostego rozwiązania. Jeżeli $n = 1$ zadanie staje się banalne. Wystarczy przełożyć ten jedyny krążek z kołka A bezpośrednio na kołek B. W przypadku dwóch krążków ($n = 2$) najpierw mniejszy krążek należy przenieść z kołka A na pomocniczy kołek C. Następnie większy krążek powinien zostać przeniesiony z A na B i wreszcie później można przełożyć krążek mniejszy z C na B. Gdy liczba krążków n jest większa, rozwiązanie zadania przypomina rozwiązanie z dwoma krążkami. Najpierw $n - 1$ krążków należy przenieść z kołka A na C traktując kołek B jako pomocniczy. Na kołku A będzie wtedy tylko największy krążek, wszystkie pozostałe będą ułożone na kołku C (jak ilustruje to rysunek 4).

Następnie największy krążek musi zostać przeniesiony bezpośrednio z kołka A na B i wreszcie później trzeba wszystkie pozostałe krążki z kołka C przenieść na kołek B, traktując kołek A jako pomocniczy. Rozwiązanie to przedstawiono na listingu 10.



Rysunek 4. Pośredni etap realizacji zadania o wieży Hanoi dla 6 krążków

```
void hanoi(int n, char A, char B, char C)
{
  // przekłada n krążków z A na B korzystając z pomocniczego kołka C
  if (n > 0)
  {
    hanoi(n-1, A, C, B);
    printf( "%c -> %c\n", A, B );
    hanoi(n-1, C, B, A);
  }
}
```

Listing 10. Rozwiązanie zadania wieży Hanoi

Éduard Lucas do swojego zadania dołączył dalekowschodnią legendę, zgodnie z którą w jednym z klasztorów buddyjskich mnisi nieustannie przekładają 64 krążki i gdy skończą wykonywanie tego zadania, nastąpi koniec świata. Jeśli przyjąć, że przełożenie pojedynczego krążka zajmuje 1 sekundę, wykonanie całego zadania będzie trwać $2^{64} - 1$ sekund co daje około 584 542 miliardów lat [1]. Konstatacja ta może i jest pocieszająca (koniec świata jeszcze daleko), ale wskazuje na największą słabość rozwiązań rekurencyjnych, jaką bardzo często jest złożoność obliczeniowa, powodująca drastyczny wzrost czasu obliczeń wraz ze wzrostem rozmiaru zadania.

6. Podsumowanie

W artykule podjęto próbę pokazania czym jest rekurencja i w jaki sposób posługiwać się nią prawidłowo. Wskazano na zalety tej metody, jak chociażby prostota wyrażania rozwiązania, ale uwypuklono także słabe strony, a w szczególności zwrócono uwagę na wysoką złożoność obliczeniową wielu rozwiązań rekurencyjnych. Warto raz jeszcze podkreślić, że rozwiązania rekurencyjne, mimo swej prostoty, często prowadzą do długotrwałych obliczeń i w wielu przypadkach warto przemyśleć zasadność stosowania takich rozwiązań w praktyce.

Dużą pomocą dla zrozumienia istoty rekurencji będzie samodzielne rozwiązywanie zadań z tego zakresu, do czego autor zachęca Czytelników. Wiele ciekawych propozycji można znaleźć na przykład w drugim rozdziale książki Sedgewicka i Wayne'a [2]. Co prawda autorzy posługują się tam językiem Java a nie, jak w niniejszym opracowaniu, językiem C, ale nie powinno to stanowić znaczącego utrudnienia.

Literatura

1. A. M. Hinz, S. Klavžar, U. Milutinović, C. Petr, *The Tower of Hanoi – Myths and Maths*, Birkhäuser, 2013.
2. R. Sedgewick, K. Wayne, *Introduction to Programming in Java: An Interdisciplinary Approach, 2nd Edition*, Addison-Wesley, 2017.