17. Iteracja a rekurencja

Do rozwiązywania wielu zagadnień algorytmicznych stosuje się iterację, czyli instrukcję polegającą na powtarzaniu wskazanych operacji. Istnieje jednak inna technika, która również wiąże się z powtarzaniem, ale nie wykorzystuje pętli. Jest nią rekurencja. W tym temacie dowiesz się, czym jest rekurencja, jakie ma zastosowania, a także dlaczego należy jej używać z rozwagą.

Cele lekcji

- Zrozumiesz zasadę działania rekurencji.
- Zaprogramujesz w języku C++ wybrane algorytmy rekurencyjne.
- Porównasz wersje iteracyjne i rekurencyjne wybranych algorytmów.
- Poznasz zasady poprawnego konstruowania algorytmów rekurencyjnych.
- Dowiesz się, czym się charakteryzują liczby Fibonacciego, oraz poznasz zasadę złotego podziału.
- Zrozumiesz rozszerzony algorytm Euklidesa.

Rekurencja (rekursja)

Rekurencja lub **rekursja** (ang. *recursion*) w informatyce oznacza odwołanie się w rozwiązaniu problemu do tego samego problemu, ale dla mniejszego rozmiaru danych. Innymi słowy, zakładamy, że znamy rozwiązanie problemu np. dla danych rzędu $n - 1$ i wykorzystujemy to, aby sformułować rozwiązanie dla danych rzędu n.

Nazwa „rekurencja" pochodzi od łacińskiego słowa *recurrere*, czyli „przybiec z powrotem". Kluczem do zrozumienia działania rekurencji są właśnie powroty z wywołania rekurencyjnego.

Warto wiedzieć

Efekt podobny do rekurencyjnego można zauważyć, stojąc plecami do dużego lustra i robiąc zdjęcie przednim aparatem telefonu (tzw. selfie).

17.1. Rekurencja w matematyce

Na lekcjach matematyki spotkaliście się lub spotkacie ze wzorami rekurencyjnymi, służącymi m.in. do obliczania wartości n-tego wyrazu ciągu. Rozważmy ciąg a_n dany wzorem:

$$a_n = \begin{cases} 2 & \text{dla } n = 1 \\ a_{n-1} + 3 & \text{dla } n > 1 \end{cases}$$

Pierwszy element tego ciągu – oznaczony jako a_1 – ma wartość 2. Aby wyznaczyć wartość a_2, należy znać wartość elementu a_1 i wykorzystać ją do obliczenia wartości według wzoru: $a_2 = a_1 + 3$. Wartość a_2 wynosi więc 5. Kolejne wyrazy ciągu a_n wyznacza się analogicznie.

Z drugiej strony, aby obliczyć wartość a_n dla danego n, należy wcześniej wyznaczyć wyraz a_{n-1}, ale żeby obliczyć tę wartość, należy znać wyraz a_{n-2} itd. Ciąg a_n jest **ciągiem arytmetycznym**, ponieważ różnica między wszystkimi kolejnymi jego wyrazami jest stała (w tym przypadku wynosi 3). Aby to wykazać, wystarczy skorzystać ze wzoru dla każdego n i sprawdzić różnicę między a_n a a_{n-1}.

Ciąg arytmetyczny

Zwróć uwagę, że odwoływanie się do poprzednich wyrazów ciągu w celu obliczenia wartości dla danego n powinno się kiedyś skończyć. We wzorze określona jest wartość dla pierwszego elementu ciągu ($n = 1$) – w informatyce nazywa się to **warunkiem początkowym**. Dzięki niemu mamy pewność, że rekurencja się zakończy.

Wartości wyrazów ciągów zdefiniowanych rekurencyjnie można łatwo obliczyć w arkuszu kalkulacyjnym. Dla powyższego ciągu wystarczy wpisać w komórce A1 wartość wyrazu początkowego, czyli 2, a w komórce A2 – formułę opisującą zależność rekurencyjną: =A1+3, i skopiować formułę w dół. Efekt widać na rysunku 17.1.

	A
1	2
2	=A1+3
3	8
4	11
5	14
6	17
7	20
8	23
9	26

Rys. 17.1. Ciąg rekurencyjny w arkuszu kalkulacyjnym

Warunek początkowy

Ćwiczenie 1

a. Zapisz wzór ogólny ciągu a_n.

b. Zapisz wzór rekurencyjny ciągu o początkowych wyrazach: $b_1 = 2$, $b_2 = 7$, $b_3 = 22$, $b_4 = 67$, w którym każdy wyraz zależy od wyrazu poprzedniego. Zastosuj ten wzór do wyznaczenia wartości b_7.

Obliczanie silni

Silnię liczby całkowitej nieujemnej oznaczamy symbolem $n!$ i opisujemy następującym wzorem:

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (n-1) \cdot n$$

Silnia

Silnię możemy łatwo zapisać w sposób rekurencyjny:

$$n! = \begin{cases} 1 & \text{dla } n < 2 \\ (n-1)! \cdot n & \text{dla } n \geqslant 2 \end{cases}$$

Warto wiedzieć

Choć może się to wydawać nienaturalne, przyjmuje się, że $0! = 1$.

Zapiszmy w pseudokodzie funkcję obliczającą rekurencyjnie wartość $n!$.

Specyfikacja

Dane: n – liczba całkowita, $0 \leqslant n \leqslant 20$.

Wynik: $n!$

Dobra rada

Wartość silni bardzo szybko rośnie. Dlatego w programach, w których wykorzystujesz silnię, ogranicz zakres danych.

```
funkcja SilniaRek(n)
    jeśli n < 2 zwróć 1 i zakończ
    w przeciwnym przypadku zwróć SilniaRek(n–1) * n i zakończ
```

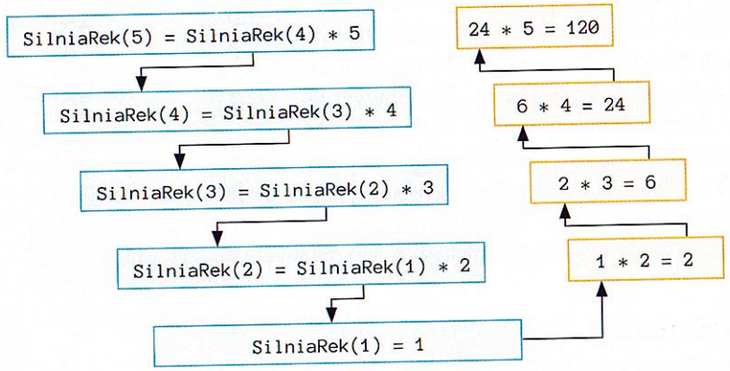
Zwróć uwagę, że powyższa funkcja wywołuje samą siebie z inną wartością parametru. Takie funkcje nazywamy **funkcjami rekurencyjnymi**.

Funkcja rekurencyjna

W językach programowania rekurencja jest realizowana właśnie z wykorzystaniem funkcji. Gdyby zabrakło sprawdzenia warunku początkowego (pierwsza linia bloku instrukcji funkcji w pseudokodzie), mielibyśmy do czynienia z **rekurencją nieskończoną**. Funkcja wywoływałaby samą siebie w nieskończoność.

Rekurencja nieskończona

Rysunek 17.2 przedstawia sposób obliczenia wartości 5! z wykorzystaniem funkcji `SilniaRek`.



Rys. 17.2. Graficzna interpretacja obliczania wartości 5! z wykorzystaniem funkcji rekurencyjnej

Wartość 5! zostanie obliczona po wywołaniu funkcji `SilniaRek(5)`. Ponieważ parametr nie jest mniejszy od 2, zostanie wywołana kopia funkcji z parametrem 4. Takie wywołania będą powtarzane do momentu wykonania funkcji z parametrem 1 (warunek początkowy), którego wynikiem będzie 1. Obliczona wartość jest zwracana do miejsca wywołania, a więc do kopii wywołania funkcji z parametrem 2, wynik pomnożony przez 2 wraca do kopii wywołania funkcji z parametrem 3 itd., aż do wywołania funkcji z parametrem 5. Po pomnożeniu przez 5 otrzymujemy końcową wartość funkcji. W pomarańczowych prostokątach znajdują się wartości generowane przy powrotach z rekurencji.

A to ciekawe

Efekt Droste – rekurencja w projektowaniu

Około 1900 r. na puszce kakao produkowanego przez holenderską firmę cukierniczą Droste pojawił się wizerunek pielęgniarki, która trzymała w ręku tacę z filiżanką oraz puszką kakao Droste. Na tej puszce widać pielęgniarkę trzymającą w ręku tacę z filiżanką oraz puszką kakao Droste itd. Ten sposób prezentacji grafiki został pod koniec lat 70. XX w. nazwany efektem Droste. Jest to rodzaj rekurencyjnego obrazu. Również dziś można zaobserwować ten zabieg na opakowaniach produktów spożywczych, m.in. amerykańskiego proszku do pieczenia i polskiego mleka zagęszczonego.

Ćwiczenie 2

Przygotuj arkusz kalkulacyjny obliczający wartości silni dla kolejnych liczb całkowitych nieujemnych mniejszych od 15.

Oto kod źródłowy programu, który oblicza wartość silni dla podanej wcześniej specyfikacji z wykorzystaniem funkcji rekurencyjnej. Wartością funkcji jest liczba **typu `long long`**. Typ ten pozwala obsługiwać liczby całkowite o dużych wartościach.

Typ `long long`, s. 328

Kod źródłowy programu obliczającego rekurencyjnie silnię

```
1.  #include<iostream>
2.
3.  using namespace std;
4.
5.  long long SilniaRek(short n)
6.  {
7.      if (n<2) return 1;
8.      return SilniaRek(n-1)*n;
9.  }
10.
11. int main ()
12. {
13.     short n;
14.     cout<<"n = "; cin>>n;
15.     cout<<n<<"! = "<<SilniaRek(n);
16.     return 0;
17. }
```

Rodzaj rekurencji, który zastosowano w powyższych przykładach do obliczania wartości według wzorów matematycznych, nazywa się **rekurencją ogonową** (ang. *tail call*). Występuje w niej dokładnie jedno wywołanie rekurencyjne – jest to ostatnia instrukcja funkcji.

Rekurencja ogonowa

Zapiszmy teraz w pseudokodzie funkcję `SilniaIter`, obliczającą iteracyjnie wartość *n*!.

```
funkcja SilniaIter(n)
    silnia ← 1
    dla i ← 2, 3, ..., n wykonuj
        silnia ← silnia * i
    zwróć silnia i zakończ
```

Warto wiedzieć

W praktyce rzadko spotyka się algorytmy wykorzystujące rekurencję ogonową. Zastępuje się ją iteracją.

Podobnie jak silnię, każdy algorytm wykorzystujący rekurencję ogonową można zapisać iteracyjnie za pomocą jednej pętli.

Ćwiczenie 3

Zapisz w języku C++ program obliczający iteracyjnie wartość *n*!.

17.2. Podział liczby na cyfry z wykorzystaniem rekurencji

Warto wiedzieć

W algorytmach wyodrębniania cyfr oraz szybkiego podnoszenia do potęgi bez wykorzystania rekurencji cyfry danej liczby przeglądane są od prawej do lewej.

Do tej pory stosowaliśmy algorytmy, które przeglądały cyfry liczby od najmniej znaczącej. Napiszemy teraz program, który z wykorzystaniem rekurencji wypisze cyfry liczby dziesiętnej od lewej strony do prawej, czyli od cyfry najbardziej znaczącej do cyfry jedności. Program nie będzie zapamiętywać cyfr w zmiennej pomocniczej.

Sformułujmy warunek zakończenia algorytmu (czyli warunek początkowy): jeśli liczba jest mniejsza od 10, to program powinien ją po prostu wypisać. Jeśli liczba jest większa niż 10, to można rekurencyjnie rozwiązać problem dla liczby podzielonej całkowicie przez 10 (czyli bez ostatniej cyfry). W ten sposób wypiszemy wszystkie cyfry liczby z wyjątkiem cyfry jedności. Następnie należy wypisać tę cyfrę, a więc resztę z dzielenia liczby przez 10. Oto specyfikacja i pseudokod:

Specyfikacja

Dane: n – liczba całkowita, $n \geqslant 0$.

Wynik: cyfry liczby n wypisane od najbardziej znaczącej do cyfry jedności, czyli od lewej do prawej.

```
funkcja CyfryRek(n)
    jeśli n < 10 to wypisz n
    w przeciwnym przypadku
        CyfryRek(n div 10)
        wypisz n mod 10
zakończ
```

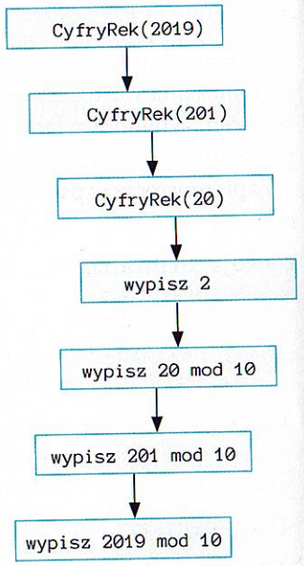
Zwróć uwagę, że wypisywanie cyfr następuje po wywołaniu rekurencyjnym, a więc w trakcie powrotów.

Wywołania rekurencyjne funkcji `CyfryRek` prześledzimy na przykładzie liczby 2019. Wartości pośrednie: 2019, 201, 20 i 2 nie są pamiętane jawnie w zmiennej – powyższa funkcja nie korzysta z żadnej zmiennej. Wartości te jednak muszą być gdzieś przechowywane. Zapewnia to mechanizm rekurencji.

Rysunek 17.3 pokazuje układ kolejnych wywołań funkcji rekurencyjnej `CyfryRek`.

Dla każdego wywołania funkcji pamiętane są aktualne wartości parametrów oraz tzw. **adres powrotu**, który określa, gdzie należy powrócić po zakończeniu danego wywołania rekurencyjnego.

Adres powrotu



Rys. 17.3. Funkcja wypisująca cyfry od najbardziej znaczących

Oto kod źródłowy funkcji `CyfryRek`:

Kod źródłowy funkcji wyznaczającej rekurencyjnie cyfry liczby od najbardziej znaczącej

```
1. void CyfryRek(int n)
2. {
3.     if (n<10) cout<<n<<endl;
4.     else
5.     {
6.         CyfryRek(n/10);
7.         cout<<n%10<<endl;
8.     }
9. }
```

Zauważ, że zapis warunku początkowego znajduje się w linii 3 kodu źródłowego i jest sprawdzany na samym początku działania funkcji.

Ćwiczenie 4

Napisz funkcję rekurencyjną `void DecToBinRek(int n)`, wypisującą cyfry binarne dziesiętnej liczby całkowitej nieujemnej od najbardziej znaczącej do najmniej znaczącej, czyli od lewej do prawej.

Zapamiętaj

Stosowanie rekurencji powoduje duże zużycie pamięci komputera. Im więcej zagłębień rekurencyjnych, tym większe obciążenie pamięci. Dlatego jeśli można rozwiązać dany problem iteracyjnie, bez wykorzystywania dodatkowych struktur danych, warto to zrobić.

17.3. Rekurencyjny algorytm szybkiego podnoszenia do potęgi

W temacie 3 omówiliśmy **iteracyjny algorytm szybkiego podnoszenia do potęgi**, wykorzystujący rozwinięcie binarne wykładnika.

Algorytm szybkiego podnoszenia do potęgi, s. 57–58

Na przykład:

$$x^{13} = x^{1101_2} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1$$

Żeby wyznaczyć wartość x^{13}, wystarczyło obliczyć kolejno:

$$x^2 = x \cdot x,\ x^4 = x^2 \cdot x^2,\ x^8 = x^4 \cdot x^4$$

Łącznie wykonaliśmy pięć mnożeń. Wartość x^{13} możemy też zapisać następująco:

$$x^{13} = (x^6)^2 x = ((x^3)^2)^2 x = ((x^2 x)^2)^2 x$$

Ćwiczenie 5

Wyznacz liczbę operacji mnożenia w ostatniej równości powyższego wzoru.

Jeśli wykładnik potęgi jest parzysty, to obliczamy potęgę o wykładniku podzielonym przez 2 i podnosimy ją do kwadratu. Dla wykładnika nieparzystego dodatkowo mnożymy wynik przez podstawę potęgi.

Ten sposób postępowania można wyrazić wzorem rekurencyjnym:

$$x^n = \begin{cases} x & \text{dla } n = 1 \\ (x^{n \text{ div } 2})^2 & \text{dla } n \text{ parzystego} \geqslant 2 \\ x \cdot (x^{n \text{ div } 2})^2 & \text{dla } n \text{ nieparzystego} \geqslant 3 \end{cases}$$

Wzór ten możemy zapisać wprost jako funkcję w pseudokodzie:

Specyfikacja

Dane: n – liczba całkowita, $n \geqslant 1$, x – liczba całkowita.

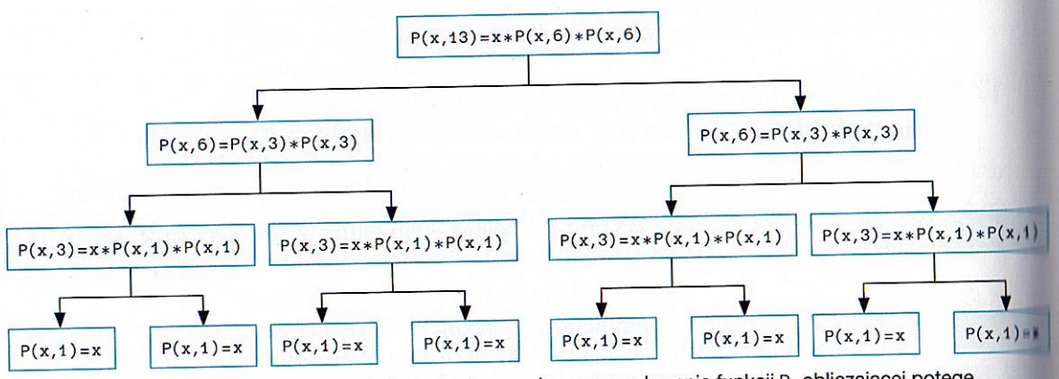
Wynik: x^n

Warto wiedzieć

Przedstawiony algorytm podnoszenia do potęgi można zastosować także wtedy, gdy podstawa jest liczbą rzeczywistą.

```
funkcja PotegaRek(x,n)
    jeśli n = 1 to zwróć x i zakończ
    jeśli n mod 2 = 0 to
        zwróć PotegaRek(x,n div 2)*PotegaRek(x,n div 2)
        i zakończ
    zwróć x*PotegaRek(x,n div 2)*PotegaRek(x,n div 2)
    i zakończ
```

Sprawdźmy, ile operacji mnożenia wykona powyższa funkcja dla $n = 13$. Rysunek 17.4 przedstawia schemat wywołań rekurencyjnych funkcji PotegaRek (na rysunku oznaczonej skrótem P) dla potęgi o wykładniku 13.



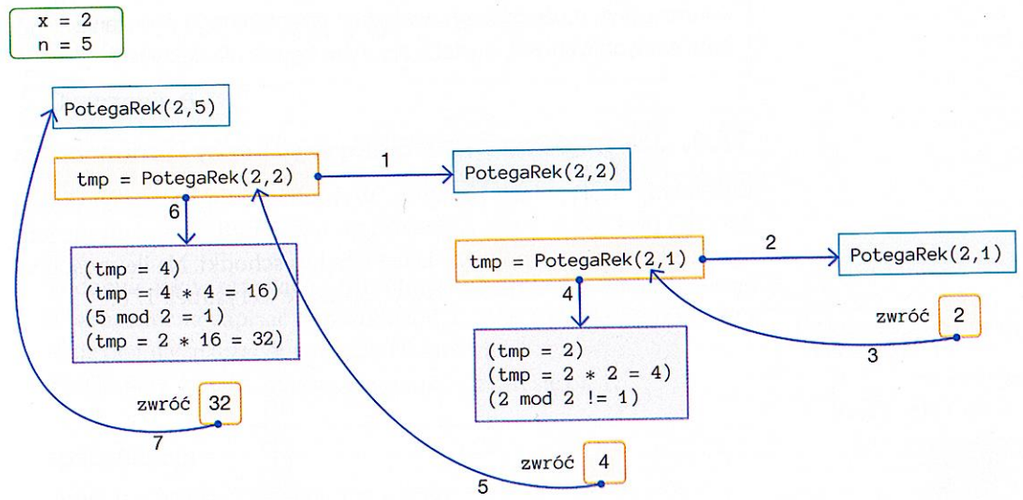
Rys. 17.4. Schemat rekurencyjnego wywoływania funkcji P, obliczającej potęgę liczby x o wykładniku 13

Zauważ, że duża liczba operacji mnożenia wynika z wielokrotnego wyliczania potęgi dla tego samego wykładnika, czyli rekurencyjnego wywoływania funkcji dla tych samych parametrów. Funkcja P zostanie wywołana dla parametrów (x,1) aż osiem razy.

Funkcję PotegaRek można zmodyfikować w taki sposób, aby zapamiętywać wynik wywołania rekurencyjnego w zmiennej pomocniczej, np. o nazwie tmp. Poniżej znajduje się pseudokod, w którym wykorzystano zmienną tmp do pojedynczego obliczania powtarzających się wywołań rekurencyjnych.

```
funkcja PotegaRek(x,n)
    jeśli n = 1 to zwróć x i zakończ
    tmp ← PotegaRek(x,n div 2)
    tmp ← tmp * tmp
    jeśli n mod 2 = 1 to tmp ← x * tmp
    zwróć tmp i zakończ
```

Rysunek 17.5 pokazuje, w jaki sposób realizowane są kolejne wywołania rekurencyjne oraz jak wykorzystywane są wartości obliczane przy powrotach z rekurencji. Funkcja PotegaRek została użyta do obliczenia wartości 2^5. Numery przy strzałkach oznaczają kolejność wywołań rekurencyjnych i obliczeń.



Rys. 17.5. Schemat kolejnych wywołań rekurencyjnych funkcji PotegaRek

Oto kod źródłowy funkcji PotegaRek:

Kod źródłowy funkcji obliczającej potęgę liczby

```
1. long long PotegaRek(int x, int n)
2. {
3.     if (n==1) return x;
4.     long long tmp=PotegaRek(x,n/2);
5.     tmp=tmp*tmp;
6.     if (n%2==1) tmp=x*tmp;
7.     return tmp;
8. }
```

Typ `long long`, s. 328

Zwróć uwagę na użyte typy danych. Ponieważ wartość potęgi szybko rośnie, został użyty **typ `long long`**. W linii 4 następuje wywołanie rekurencyjne funkcji `PotegaRek` z parametrami `x` oraz `n/2`, której wynik jest przypisywany zmiennej `tmp`.

W wersji rekurencyjnej algorytmu przyjęliśmy, że $n \geqslant 1$. Dla $n = 0$ funkcja nie będzie działać prawidłowo – nastąpią nieskończone wywołania rekurencyjne i w efekcie pojawi się błąd – przekroczenie pamięci dostępnej dla programu. Warunek końca można zastąpić warunkiem:

```
jeśli n = 0 to zwróć 1 i zakończ
```

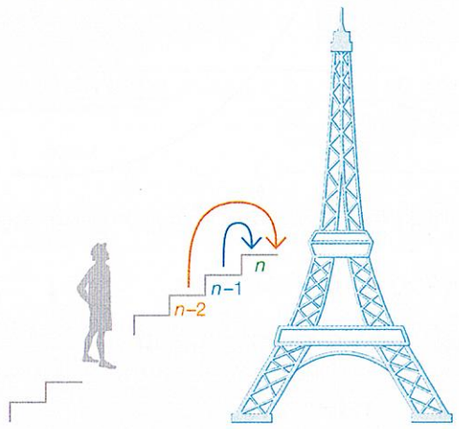
Spowoduje to jednak wykonywanie dodatkowych operacji mnożenia przy obliczaniu każdej potęgi (podnoszenie 1 do kwadratu i mnożenie przez x). W takiej sytuacji warto ten szczególny przypadek rozpatrywać poza funkcją rekurencyjną.

Zapamiętaj

Jeśli wzór zdefiniowany rekurencyjnie zamienimy wprost na funkcję rekurencyjną, może to doprowadzić do wielokrotnego wyliczania tych samych wartości, a wtedy algorytm będzie nieefektywny.

17.4. Jak wejść na wieżę Eiffla, czyli liczby Fibonacciego

Rozważmy następujący problem. Wybraliście się na wycieczkę do Paryża i próbujecie wejść schodami na wieżę Eiffla. Załóżmy, że jednym krokiem można pokonać jeden lub dwa schodki. Na ile sposobów da się wejść na n-ty schodek wieży? Pokazuje to rysunek 17.6.



Rys. 17.6. Sposób wejścia na n-ty schodek wieży

Istnieje tylko jeden sposób wejścia na pierwszy schodek wieży. Na drugi schodek można wejść już na dwa sposoby: trzeba zrobić dwa kroki po jednym schodku lub jeden krok przez dwa schodki.

Warto wiedzieć

Leonardo z Pizy (1175–1250), zwany Fibonaccim, był włoskim matematykiem. Opracował wiele dzieł, ale do najważniejszych należy traktat *Liber abaci* (*Księga liczydła* lub *Księga rachunków*), w którym opisał system pozycyjny liczb oraz podstawy arytmetyki.

Na n-ty schodek można wejść ze schodka $n - 1$ (jednym krokiem pokonujemy jeden schodek) lub ze schodka $n - 2$ (jednym krokiem pokonujemy dwa schodki). Zatem możliwości wejścia na n-ty schodek jest tyle, ile jest łącznie możliwości wejścia na $n - 1$ oraz $n - 2$ schodek.

Możemy to zapisać wzorem rekurencyjnym:

$$s_n = \begin{cases} 1 & \text{dla } n = 1 \\ 2 & \text{dla } n = 2 \\ s_{n-1} + s_{n-2} & \text{dla } n > 2 \end{cases}$$

W 1202 r. włoski matematyk Leonardo z Pizy, zwany Fibonaccim, w swoim dziele *Liber abaci* opisał następujący ciąg liczbowy: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..., w którym każdy kolejny wyraz ciągu jest sumą dwóch poprzednich. Ciąg ten nazywany jest **ciągiem Fibonacciego** ○ Ciąg Fibonacciego i określony jest wzorem:

$$f_n = \begin{cases} 0 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ f_{n-1} + f_{n-2} & \text{dla } n > 1 \end{cases}$$

Ćwiczenie 6

a. Porównaj ciągi s_n i f_n. Jaka jest zależność między nimi?
b. Przygotuj arkusz kalkulacyjny obliczający wartości kolejnych liczb Fibonacciego.

Podobnie jak w przypadku innych wzorów rekurencyjnych, powyższy wzór można zapisać w pseudokodzie jako funkcję rekurencyjną. Będzie ona wyznaczać n-tą liczbę Fibonacciego.

Przyjmijmy, że ciąg Fibonacciego rozpoczyna się od 1.

Specyfikacja

Dane: n – liczba całkowita, $1 \leqslant n \leqslant 50$.

Wynik: f_n – n-ta liczba Fibonacciego: $f_n = \begin{cases} 1 & \text{dla } n < 3 \\ f_{n-1} + f_{n-2} & \text{dla } n \geqslant 3 \end{cases}$

```
funkcja FibRek(n)
    jeśli n < 3 to zwróć 1 i zakończ
    zwróć FibRek(n–1) + FibRek(n–2) i zakończ
```

Ćwiczenie 7

Napisz program według powyższej specyfikacji z wykorzystaniem funkcji `FibRek`. Sprawdź działanie programu dla liczb: 5, 10 i 45.

Warto wiedzieć

Ciąg Fibonacciego często definiuje się od 1:

$$f_n = \begin{cases} 1 & \text{dla } n = 1 \text{ lub } n = 2 \\ f_{n-1} + f_{n-2} & \text{dla } n \geqslant 3 \end{cases}$$

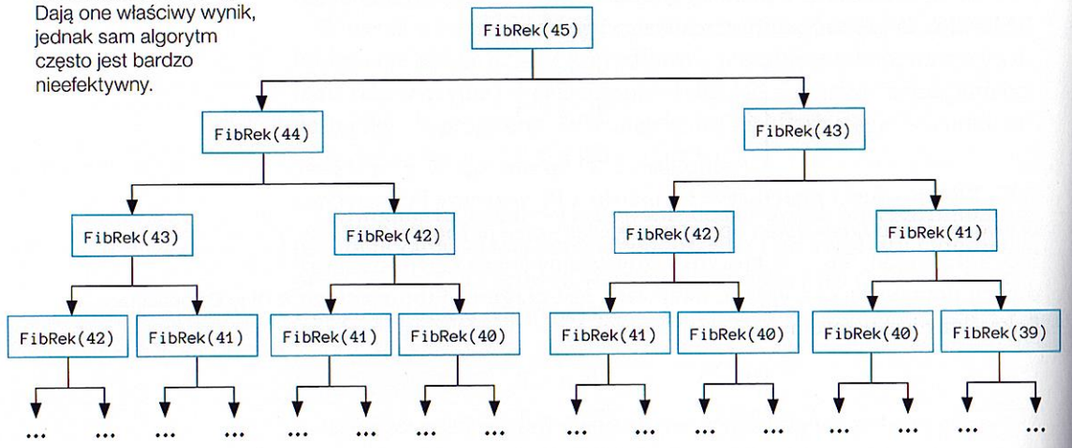
Dobra rada

Liczby w ciągu Fibonacciego rosną bardzo szybko. Dlatego w programach wykorzystujących ten ciąg ogranicz zakres danych.

Dobra rada

Unikaj przepisywania wprost rekurencyjnych wzorów matematycznych. Dają one właściwy wynik, jednak sam algorytm często jest bardzo nieefektywny.

Zauważ, że oczekiwanie na 45. liczbę Fibonacciego trwa długo. Przyczyna, podobnie jak w przykładzie z obliczaniem potęgi, leży w wielokrotnym wyznaczaniu tych samych liczb Fibonacciego (rys. 17.7).



Rys. 17.7. Schemat wywołań funkcji `FibRek` dla $n = 45$

Ćwiczenie 8

a. Ile razy zostanie wywołana funkcja `FibRek` z rysunku 17.7 dla kolejnych liczb mniejszych od 45?

b. Ustal, ile operacji dodawania wystarczy wykonać, aby wyznaczyć n-tą liczbę Fibonacciego ręcznie na kartce papieru lub w arkuszu kalkulacyjnym.

Liczby Fibonacciego – wersja iteracyjna algorytmu

Zauważ, że aby wyznaczyć kolejną liczbę Fibonacciego, wystarczy w danym momencie pamiętać dwie ostatnie liczby. Obliczanie wartości kolejnych liczb można zapisać za pomocą pojedynczej pętli. Oto zapis funkcji w pseudokodzie:

```
funkcja FibIter(n)
    f1 ← 1
    f2 ← 1
    dla i ← 3, 4, ..., n wykonuj
        tmp ← f2
        f2 ← f1 + f2
        f1 ← tmp
    zwróć f2 i zakończ
```

Warto wiedzieć

Kolejną liczbę Fibonacciego na podstawie dwóch poprzednich można także wyznaczyć bez użycia zmiennej pomocniczej. Wystarczy w tym celu wykonać dodatkowe odejmowanie. Realizuje to sekwencja instrukcji:

```
f2 ← f1 + f2
f1 ← f2 - f1
```

W zmiennych `f1` i `f2` pamiętane są dwie ostatnie liczby Fibonacciego na danym etapie obliczeń. Kiedy program wyznacza kolejną liczbę, w zmiennej pomocniczej `tmp` przechowywana jest aktualnie ostatnia liczba, która w kolejnym kroku będzie liczbą przedostatnią.

W zmiennej `f2` wyliczana jest wartość nowej liczby jako suma dwóch ostatnich. Dotychczasowa wartość `f1` nie jest już potrzebna, ponieważ jest to w danym momencie trzecia liczba od końca. Dlatego zmienna `f1` przyjmuje wartość przedostatniej liczby pamiętanej tymczasowo w zmiennej `tmp`.

Oto kod źródłowy programu wyznaczającego n-ty wyraz ciągu Fibonacciego z wykorzystaniem funkcji `FibIter`:

Kod źródłowy programu wyznaczającego *n*-ty wyraz ciągu Fibonacciego

```
1.  #include<iostream>
2.
3.  using namespace std;
4.
5.  long long FibIter(short n)
6.  {
7.      long long f1=1, f2=1, tmp;
8.      for (int i=3;i<=n;i++)
9.          {
10.             tmp=f2;
11.             f2=f1+f2;
12.             f1=tmp;
13.         }
14.     return f2;
15. }
16.
17. int main ()
18. {
19.     short n;
20.     cout<<"n = "; cin>>n;
21.     cout<<"f("<<n<<") = "<<FibIter(n);
22.     return 0;
23. }
```

W nagłówku funkcji `FibIter` użyto typu **`long long`**, ponieważ wynikiem działania funkcji może być duża liczba całkowita. Parametrami funkcji będą stosunkowo małe liczby, dlatego można w ich wypadku zastosować **typ `short`**.

Typ **short**, s. 328

Ćwiczenie 9

Przygotuj arkusz kalkulacyjny, który obliczy ilorazy kolejnych liczb Fibonacciego $\frac{f_n}{f_{n-1}}$. Do jakiej wartości zbliżają się kolejne ilorazy?

Fenomen liczb Fibonacciego polega na tym, że bardzo często występują wokół nas. Odnajdziemy je w przyrodzie, architekturze i sztuce. Ciekawym zagadnieniem związanym z liczbami Fibonacciego jest tzw. **złota proporcja**, o której możesz przeczytać na s. 248–249.

Złota proporcja

17.5. Rozszerzony algorytm Euklidesa

Algorytm Euklidesa, s. 102

Poprawność **algorytmu Euklidesa** wykazywaliśmy na podstawie zależności rekurencyjnej: $NWD(a, b) = NWD(a - b, b)$ przy założeniu, że $a > b$.

Obliczanie NWD można zatem wyrazić prostym wzorem rekurencyjnym, który ma zastosowanie zarówno w wersji algorytmu z odejmowaniem, jak i z dzieleniem.

$$NWD(a,b) = \begin{cases} a & \text{dla } a = b \\ NWD(a-b,b) & \text{dla } a > b \\ NWD(a,b-a) & \text{dla } b > a \end{cases}$$

$$NWD(a,b) = \begin{cases} a+b & \text{dla } a = 0 \text{ lub } b = 0 \\ NWD(a \bmod b, b) & \text{dla } a > b \\ NWD(a, b \bmod a) & \text{dla } b > a \end{cases}$$

Powyższy wzór można zapisać prościej:

$$NWD(a,b) = \begin{cases} a & \text{dla } b = 0 \\ NWD(b, a \bmod b) & \text{dla } b > 0 \end{cases}$$

Zauważ, że większa z dwóch liczb jest podczas obliczeń pamiętana w zmiennej a. Jeśli na początku a nie jest większą liczbą, pierwsze wywołanie rekurencyjne będzie polegało na zamianie wartości zmiennych a i b.

Warto wiedzieć

Jeśli $a < b$ i $b \neq 0$, to:
$NWD(a,b) =$
$= NWD(b, a \bmod b) =$
$= NWD(b,a)$

Funkcję rekurencyjną obliczającą największy wspólny dzielnik według powyższego wzoru można zapisać następująco:

```
funkcja NWDRek(a,b)
    jeśli b = 0 to zwróć a i zakończ
    zwróć NWDRek(b,a mod b) i zakończ
```

Ćwiczenie 10

Zapisz w języku C++ funkcję NWDRek i sprawdź jej działanie.

Rozszerzony algorytm Euklidesa

W niektórych zastosowaniach informatycznych, np. w kryptografii, wykorzystuje się **rozszerzony algorytm Euklidesa**, który największy wspólny dzielnik pozwala wyrazić wzorem:

$$NWD(a, b) = x \cdot a + y \cdot b$$

gdzie x i y są liczbami całkowitymi (mogą być także ujemne).

Poszukujemy zatem nie jednej liczby, ale pary liczb (x, y) spełniającej powyższe równanie dla danych liczb całkowitych a i b. Oznacza to, że algorytm nie tylko wyznacza NWD podanych liczb a i b, lecz także przedstawia ten dzielnik w postaci sumy liczb a i b pomnożonych przez współczynniki – odpowiednio x i y. Zapiszemy teraz specyfikację tak sformułowanego problemu.

Specyfikacja

Dane: a, b – liczby całkowite, $a > 0$, $b > 0$.

Wynik: x, y – liczby całkowite spełniające równanie:
$NWD(a, b) = x \cdot a + y \cdot b$.

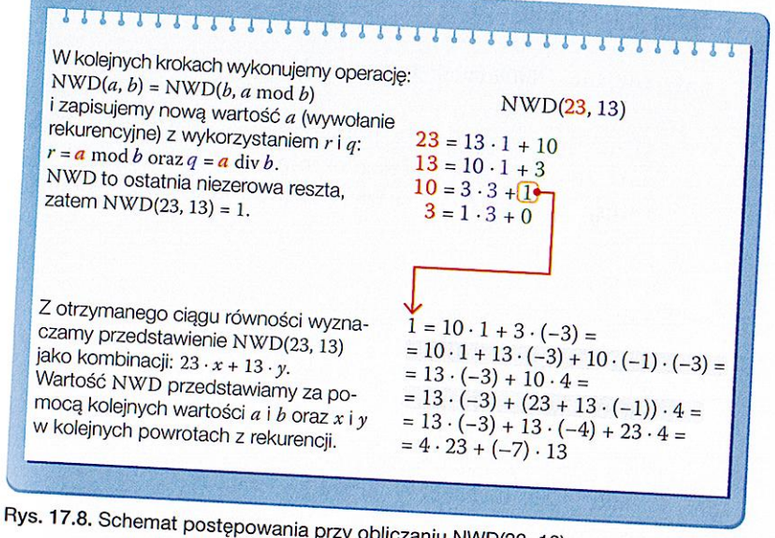
Algorytm wyznaczania współczynników x i y będzie polegał na modyfikacji funkcji NWDRek i wykorzystaniu prostych zależności arytmetycznych. Oznaczmy przez q wynik dzielenia całkowitego a przez b, a przez r – resztę z tego dzielenia:

$$q = a \text{ div } b$$
$$r = a \bmod b$$

Prześledźmy, jak się zmieniają wartości a, b, q i r (oznaczono je kolorami) podczas obliczania wartości NWD(23, 13) oraz jak wyznaczane są wartości x i y (rys. 17.8).

Warto wiedzieć

$NWD(a,b)$ zawsze można przedstawić w postaci $x \cdot a + y \cdot b$.
Nie jest to jedyne przedstawienie, ponieważ dla dowolnej liczby całkowitej k poprawne jest również przedstawienie:
$(x - k \cdot b) \cdot a + (y + k \cdot a) \cdot b$.



Rys. 17.8. Schemat postępowania przy obliczaniu NWD(23, 13)

W górnej części schematu znajdują się kolejne wywołania rekurencyjne, a poniżej zostały rozpisane kolejne przedstawienia liczby 1 z wykorzystaniem aktualnych wartości a i b oraz x i y. W ostatnim kroku otrzymujemy przedstawienie dla wyjściowych wartości a i b:

$$NWD(23, 13) = 4 \cdot 23 + (-7) \cdot 13$$

Zatem poszukiwaną parą liczb jest $(4, -7)$.

Zauważ, że w każdym kroku wartość x jest równa wartości y z poprzedniego kroku – oznaczmy ją przez y'. Wartość y jest równa $x' + (-q) \cdot y'$, gdzie x' to wartość x z poprzedniego kroku. Widać to na rysunku 17.8 w równościach zaznaczonych kolorem niebieskim.

Warto wiedzieć

Aby reprezentować dwie liczby całkowite, można zdefiniować własny typ danych. Może nim być struktura złożona z dwóch pól typu całkowitego. Oto definicja takiej struktury:

```
struct wspolczynniki
{
    int x,y;
};
```

Typ pair, s. 332

Aby zapisać algorytm rekurencyjny, za warunek początkowy przyjmiemy stan, w którym b będzie równe 0.

Możemy już zapisać w pseudokodzie funkcję NWDRozRek, obliczającą wartości x i y.

```
funkcja NWDRozRek(a,b)
    jeśli b = 0 to zwróć (1,0) i zakończ
    (x,y) ← NWDRozRek(b,a mod b)
    zwróć (y,x-(a div b) * y) i zakończ
```

Wartością funkcji jest para liczb całkowitych. Należy zatem użyć typu danych, który reprezentuje dwie liczby całkowite. Wygodnie będzie wykorzystać **typ pair** z biblioteki STL.

Oto kod źródłowy programu realizującego rozszerzony algorytm Euklidesa z wykorzystaniem funkcji NWDRozRek:

Kod źródłowy programu realizującego rozszerzony algorytm Euklidesa

```
1.  #include<iostream>
2.
3.  using namespace std;
4.
5.  pair<int,int> NWDRozRek(int a, int b)
6.  {
7.      if (b==0) return make_pair(1,0);
8.      pair<int,int> pom=NWDRozRek(b,a%b);
9.      return make_pair(pom.second,pom.first-(a/b)*pom.second);
10. }
11.
12. int main ()
13. {
14.     int a,b;
15.     cout<<"a="; cin>>a;
16.     cout<<"b="; cin>>b;
17.     pair<int,int> w=NWDRozRek(a,b);
18.     cout<<"x="<<w.first<<" y="<<w.second;
19.     return 0;
20. }
```

W liniach 5–10 zdefiniowana jest funkcja NWDRozRek typu **pair<int,int>**. W linii 7 tworzona jest para złożona z liczb 1 i 0, która jest wartością funkcji dla warunku początkowego. Zmienna pomocnicza pom, deklarowana jako para (linia 8), przechowuje wynik rekurencyjnego wywołania funkcji NWDRozRek. W linii 9 tworzona jest para zgodnie z zasadą tworzenia nowych wartości zmiennych x i y. Wykorzystujemy do tego celu **funkcję** make_pair. W linii 17 deklarowana jest zmienna w typu **pair<int,int>**, która przechowuje wynik działania funkcji NWDRozRek dla wczytanych danych. W linii 18 wypisywane są elementy pary w.

Funkcja make_pair, s. 332

Podsumowanie

- Rekurencja to odwołanie się w rozwiązaniu problemu do tego samego problemu, ale dla mniejszego rozmiaru danych.
- Algorytm rekurencyjny powinien mieć sformułowany warunek końca, nazywany warunkiem początkowym, dla którego nie ma już wywołania rekurencyjnego.
- Bezpośrednie przełożenie rekurencyjnych wzorów matematycznych na funkcje często prowadzi do bardzo nieefektywnych algorytmów.
- Korzystanie z rekurencji znacząco obciąża pamięć komputera.
- Jeśli dany problem można rozwiązać zarówno iteracyjnie, jak i rekurencyjnie, lepiej zrobić to iteracyjnie.
- W rekurencji ogonowej występuje dokładnie jedno wywołanie rekurencyjne – jest to ostatnia instrukcja. Można ją zastąpić algorytmem iteracyjnym z zastosowaniem pojedynczej pętli.
- Każdy algorytm rekurencyjny można sprowadzić do iteracji. Często jest to jednak skomplikowane i wymaga użycia dodatkowych struktur danych.

Zadania

* **1** Dany jest ciąg a_n, określony następującym wzorem rekurencyjnym:

$$a_n = \begin{cases} 1 & \text{dla } n = 0 \\ 2 \cdot a_{n-1} & \text{dla } n > 0 \end{cases}$$

a. Przygotuj arkusz kalkulacyjny obliczający kolejne wyrazy ciągu.

b. Jak nazywamy ciąg, w którym kolejny wyraz powstaje przez pomnożenie poprzedniego przez określoną wartość? Jakie wartości reprezentują wyrazy powyższego ciągu?

c. Napisz w pseudokodzie dwie funkcje, które będą obliczać n-ty wyraz ciągu – jedna w sposób rekurencyjny, druga – w iteracyjny.

* **2** Napisz program obliczający największy wspólny dzielnik n liczb całkowitych dodatnich. Problem rozwiąż iteracyjnie.
Uwaga: Możesz skorzystać z poniższej zależności rekurencyjnej.
$NWD(a_1, a_2, a_3, \ldots, a_n) = NWD(a_1, NWD(a_2, a_3, \ldots, a_n))$

* **3** W pliku otrzymanym od nauczyciela (np. *ciag_rekurencyjny.xlsx*) znajduje się ciąg liczbowy wraz z formułą go tworzącą. Na tej podstawie zapisz pseudokod funkcji rekurencyjnej.

* **4** Zapisz w pseudokodzie funkcję wyznaczającą n-ty wyraz ciągu, którego początkowe wyrazy mają wartości: $a_1 = -2$, $a_2 = 4$, $a_3 = -8$, $a_4 = 16$, $a_5 = -32$ i w którym wartość każdego następnego wyrazu zależy od wyrazu poprzedniego.