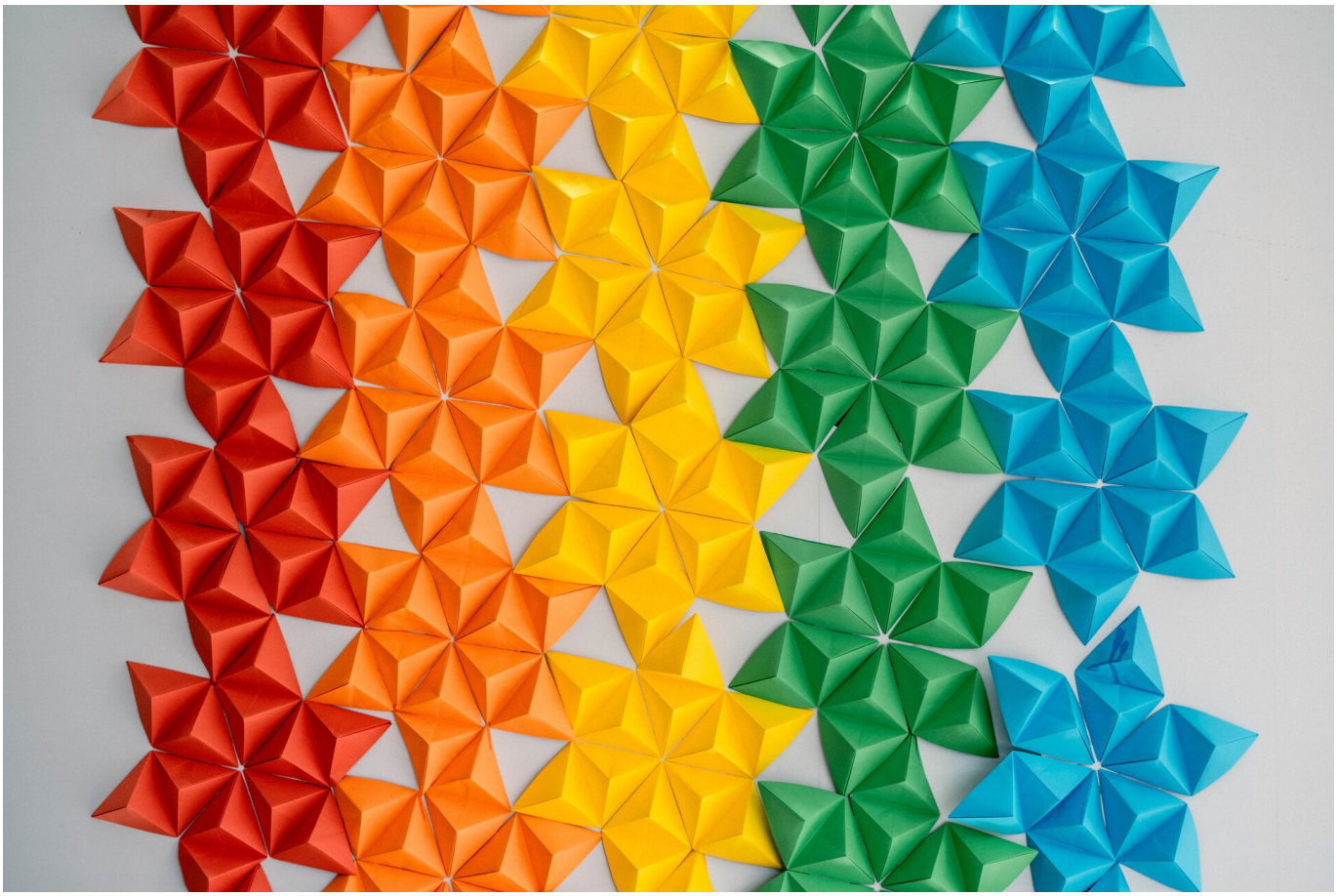


Rekurencja a iteracja w języku C++

Źródło: Chinh Le Duc, domena publiczna.

Wiesz już, na czym polega różnica między iteracją a rekurencją. W wielu sytuacjach metody te mogą być wykorzystywane zamiennie. Zdarza się jednak, że rekurencja błędnie stosowana jest w problemach, których natura nie jest rekurencyjna. Zrozumienie wad i zalet rekurencji wymaga umiejętności tworzenia algorytmów w sposób rekurencyjny, dlatego warto przećwiczyć rozwiązywanie problemów zarówno za pomocą rekurencji, jak i iteracji.

W e-materiale Rekurencja a iteracja porównujemy te dwie metody oraz omawiamy wady i zalety każdej z nich. Tutaj natomiast zajmiemy się analizą tego zagadnienia w języku C++.

Z implementacją w innych językach programowania zapoznasz się w pozostałych e-materiałach z tej serii:

- Rekurencja a iteracja w języku Java,
- Rekurencja a iteracja w języku Python.

Więcej przykładów, ćwiczeń i rozwiązań znajdziesz tu: Rekurencja a iteracja - zadania maturalne.

O tym, jak zagadnienie rekurencji wyjaśnia matematyka, dowiesz się z e-materiałów:

- Ciąg określony rekurencyjnie,
- Ciąg geometryczny określony rekurencyjnie,
- Wzór ogólny ciągu określonego rekurencyjnie,
- Ciąg arytmetyczny określony wzorem rekurencyjnym.

Twoje cele

- Podsumujesz informacje na temat podstawowych właściwości iteracji i rekurencji.
- Porównasz metodę iteracyjną i rekurencyjną w języku C++.
- Rozwiążesz zadany problem metodą rekurencyjną i iteracyjną.

Przeczytaj

Rekurencja a iteracja w praktyce

Do rozwiązania problemu możemy wykorzystać zarówno metodę iteracyjną, jak i rekurencyjną (rekursywną). Ale która z nich jest lepsza?

Nie ma tu jednoznacznej odpowiedzi. Metoda rekurencyjna wymaga wolnego miejsca w stosie. Stos używany jest do zapisu zmiennych lokalnych, przekazywanych argumentów, jak również adresu powrotu. Każde kolejne wywołanie funkcji rekurencyjnej wymaga użycia pamięci stosu dla tych danych, co sprawia, że liczba wywołań jest ograniczona. Jednak w niektórych przypadkach metoda rekurencyjna jest czytelniejsza niż iteracyjna, np. w algorytmie sortowania przez scalanie.

Zmiana rozmiaru stosu

Jeżeli wyczerpie się pamięć przeznaczona na stos, system operacyjny zgłosi błąd i zakończy działanie programu. Przepełnienie stosu sygnalizowane jest zwykle komunikatem `Segmentation fault` (w systemach Linux) lub `Access Violation` (w systemach Windows).

Rozmiar stosu ogranicza maksymalną liczbę wywołań rekurencyjnych. Domyślnie jest to rozmiar rzędu kilku megabajtów i zależy od użytego kompilatora oraz systemu operacyjnego. W systemie Linux z kompilatorem GCC wynosi on domyślnie 8 MB, co pozwala na ok. 200 000 wywołań rekurencyjnych. Rozmiar stosu można tu zmienić systemowym poleceniem `ulimit -s [wartość w kB]`. W systemie Windows o rozmiarze stosu decyduje linker, domyślnie ma on rozmiar 1 MB.

Ciąg Fibonacciego – implementacja rekurencyjna oraz iteracyjna

Przeanalizujmy powyższe rozumowanie na konkretnym przykładzie – porównajmy dwie funkcje obliczające n-ty wyraz ciągu Fibonacciego.

Na początek przypomnijmy definicję ciągu Fibonacciego:

$$F_n = \begin{cases} 0 & \text{gdy } n = 0 \\ 1 & \text{gdy } n = 1 \\ F_{n-1} + F_{n-2} & \text{gdy } n > 1 \end{cases}$$

Jest to definicja rekurencyjna, zatem możemy zaimplementować ją wprost, zgodnie z poniższą specyfikacją:

Specyfikacja:

Dane:

- `n` – liczba naturalna oznaczająca numer elementu ciągu Fibonacciego do obliczenia przez program

Wynik:

- `liczba` – liczba całkowita, n-ty wyraz ciągu Fibonacciego

```
long long fibonacciRekurencyjnie(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    return fibonacciRekurencyjnie(n-1) + fibonacciRekurencyjnie(n
}
```

Przykładową implementację iteracyjną możemy zrealizować następująco (ona również spełnia warunki powyższej specyfikacji):

```
long long fibonacciIteracyjnie(int n) {
    long long wyrazAktualny = 0,
              wyrazNastepny = 1,
              wyrazPoprzedni;

    for (int i = 0; i < n; i++) {
        wyrazPoprzedni = wyrazAktualny;
        wyrazAktualny = wyrazNastepny;
        wyrazNastepny += wyrazPoprzedni;
    }

    return wyrazAktualny;
}
```

Sprawdźmy teraz, która implementacja jest wydajniejsza. W tym celu uruchomimy prosty kod, który wykona po 20 wywołań każdej z napisanych funkcji ze stopniowo rosnącym argumentem. Aby porównać liczbę czynności koniecznych do wykonania przez każdą z funkcji, zadeklarujemy dwie zmienne globalne: `licznikRekurencyjny` oraz

`licznikIteracyjny`. Pierwsza z nich będzie zwiększana z każdym kolejnym wywołaniem funkcji rekurencyjnej, a druga - w pętli w funkcji iteracyjnej. W funkcji głównej programu porównamy wyniki, a przed każdym kolejnym porównaniem obu funkcji liczniki zostaną wyzerowane.

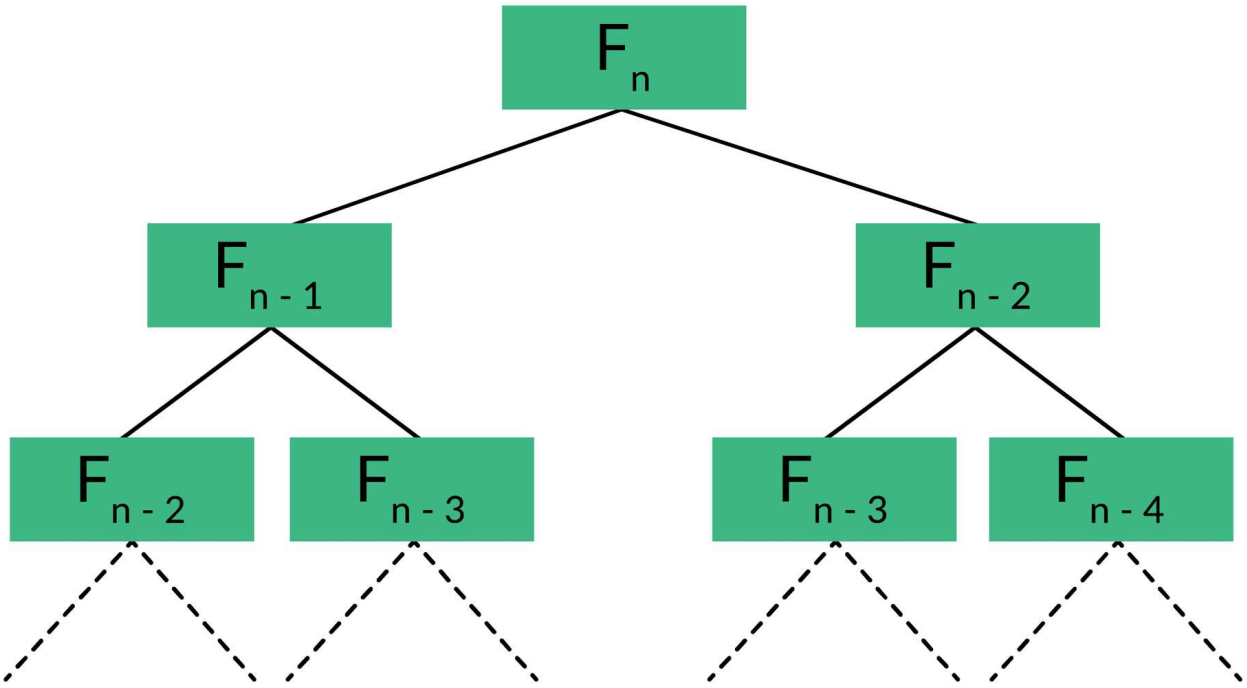
```
#include <iostream>
using namespace std;

int licznikRekurencyjny,
    licznikIteracyjny;

long long fibonacciRekurencyjnie(int n) {
    licznikRekurencyjny++;
    if (n == 0) return 0;
    else if (n == 1) return 1;
    return fibonacciRekurencyjnie(n - 1) + fibonacciRekurencyjnie
}

long long fibonacciIteracyjnie(int n) {
    long long wyrazAktualny = 0,
              wyrazNastepny = 1,
              wyrazPoprzedni;

    for (int i = 0; i < n; i++) {
        licznikIteracyjny++;
        wyrazPoprzedni = wyrazAktualny;
        wyrazAktualny = wyrazNastepny;
        wyrazNastepny += wyrazPoprzedni;
    }

    return wyrazAktualny;
}

int main() {
    cout << "Rekurencyjnie:\tIteracyjnie:" << endl;
    for (int i = 0; i < 20; ++i) {
        licznikIteracyjny = 0;
        licznikRekurencyjny = 0;

        fibonacciIteracyjnie(i);
        fibonacciRekurencyjnie(i);
```

```
37
38          cout << licznikRekurencyjny << "\t" << licznikIteracyjny
39      }
40  }
```

Po uruchomieniu kod wypisuje na standardowe wyjście programu następujące rezultaty:

```
1  Rekurencyjnie:  Iteracyjnie:
2  1       0
3  1       1
4  3       2
5  5       3
6  9       4
7  15      5
8  25      6
9  41      7
10 67      8
11 109     9
12 177     10
13 287     11
14 465     12
15 753     13
16 1219    14
17 1973    15
18 3193    16
19 5167    17
20 8361    18
21 13529   19
```

Oczywiście wyników nie należy interpretować dosłownie, gdyż nie jesteśmy w stanie stwierdzić, która operacja jest bardziej kosztowna – pojedyncze wywołanie funkcji czy pojedyncze wykonanie kodu w pętli. Widzimy jednak, że złożoność obliczeniowa funkcji rekurencyjnej jest znacząco większa niż w przypadku funkcji iteracyjnej. Przyczynę takiego stanu rzeczy przedstawia schemat (F_n oznacz n-ty wyraz ciągu Fibonacciego):



Już na drugim i trzecim poziomie wartość F_{n-2} musi zostać policzona dwa razy osobno. Podobna sytuacja powtarza się podczas kolejnych wywołań – dla F_{n-3}, F_{n-4} itd. Wielokrotnie liczymy te same wartości, przez co zwiększa się złożoność obliczeniowa.

Spróbujmy porównać również złożoność pamięciową obu funkcji. W przypadku metody iteracyjnej możemy od razu stwierdzić, że jest ona stała. Funkcja iteracyjna deklaruje tylko dwie (włącznie z parametrem) zmienne typu `int` i trzy zmienne typu `long long`. Natomiast funkcja rekurencyjna – biorąc pod uwagę fakt, że przy każdym kolejnym wywołaniu musimy zapisać na stosie przekazywane argumenty, adres powrotu oraz wszystkie zmienne lokalne – potrzebuje dużo więcej pamięci. Dodatkową wadą funkcji rekurencyjnej jest również możliwość przepełnienia stosu dla dużego argumentu `n`.

Łatwo zatem stwierdzić, że rekurencyjne obliczanie elementów ciągu Fibonacciego nie jest dobrym przykładem zastosowania rekurencji. Błędem byłoby jednak założenie, że należy unikać tej metody. Istnieją algorytmy oraz struktury danych specjalnie projektowane i optymalizowane pod rekurencję. Przykładem mogą być tu algorytmy szybkiego sortowania, np. sortowanie przez scalanie lub sortowanie szybkie (quick sort). Zadaniem programisty jest ocenienie, które podejście będzie lepsze dla danego problemu.

Silnia – implementacja rekurencyjna oraz iteracyjna

Funkcję silnia możemy zdefiniować iteracyjnie w następujący sposób:

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

Z kolei definicja rekurencyjna może wyglądać tak:

$$n! = \begin{cases} 1 & \text{gdy } n = 0 \vee n = 1 \\ n \cdot (n-1)! & \text{gdy } n > 1 \end{cases}$$

Aby lepiej wskazać różnice między obiema metodami, zaimplementujemy (korzystając z powyższych definicji) obliczanie wartości silni.

Specyfikacja:

Dane:

- `n` – liczba naturalna, dla której należy obliczyć wartość $n!$

Wynik:

- `silnia` – liczba naturalna, wartość $n!$

Obliczanie metodą rekurencyjną:

```cpp
#include <iostream>
using namespace std;

long long silniaRekurencja(int);

int main() {
  cout << silniaRekurencja(5) << endl;

  return 0;
}

long long silniaRekurencja(int n) {
    if (n == 0) {
        return 1;
    } else if (n == 1) {
        return 1;
    }

    return n * silniaRekurencja(n - 1);
}
```

Obie zasady działania rekurencji zostały zachowane.

1. Każde kolejne rekurencyjne wywołanie funkcji ma inną wartość argumentu, ponieważ za każdym razem zmienna `n` zmniejsza swoją wartość o 1;
2. Funkcja będzie wywoływana tak długo, aż zmienna `n` osiągnie wartość 1. Gdy tylko tak się stanie, funkcja zwróci wartość 1! do funkcji, która ją wywołała. Ta z kolei pomnoży tę liczbę przez wartość zmiennej `n`, która została jej przekazana, a następnie zwróci wynik. Czynność ta będzie powtarzać się do momentu, gdy końcowa wartość zostanie zwrócona do funkcji `main`, gdzie zostanie wypisana na ekran.

> **Ważne!**
>
> W sytuacji, kiedy wartość parametru wynosi 0 lub 1, należy zwrócić wartość 1.

Oto fragment kodu poszerzony o implementację algorytmu wyliczającego wartość silni metodą iteracyjną:

```cpp
#include <iostream>
using namespace std;

// Metoda rekurencyjna
long long silniaRekurencja(int n) {
    if (n == 0) {
        return 1;
    } else if (n == 1) {
        return 1;
    }

    return n * silniaRekurencja(n - 1);
}

// Metoda iteracyjna
long long silniaIteracyjna(int n) {
    long long silniaIteracyjna = 1;

    for (int i = 2; i <= n; i++) {
        silniaIteracyjna *= i;
    }

    return silniaIteracyjna;
}

int main() {
  // Argument silni
  int n;
```

```
29    cin >> n;
30
31    // Wypisywanie wyników
32    cout << "Silnia rekurencyjna: " << silniaRekurencja(n) << endl;
33    cout << "Silnia iteracyjna: " << silniaIteracyjna(n) << endl;
34
35    return 0;
36
37 }
```

W metodzie iteracyjnej zmienną sterującą pętli `for` inicjujemy wartością 2, ponieważ mnożenie $1 \cdot 1$ nie będzie miało żadnego wpływu na wynik działania programu. Przy każdej iteracji wartość `silniaIteracyjna` jest mnożona przez `i`. Pętla kończy działanie, gdy wartość zmiennej sterującej staje się większa niż zmienna `n`.

Bez względu na to, jakiej metody użyjemy, wynik będzie taki sam. Główną różnicą jest wygoda i sposób zapisu. Poza tym przy zastosowaniu metody rekurencyjnej program będzie wykonywał się nieco dłużej. Mają na to wpływ instrukcje warunkowe, które będą sprawdzane przy każdym wywołaniu funkcji, jak również zapis i odczyt ze stosu.

Słownik

> **adres powrotu**
>
> adres w pamięci, od którego program powinien kontynuować działanie po zakończeniu wywołania funkcji
> **iteracja**
>
> metoda polegająca na używaniu pętli w celu wykonania zawartego w nich kodu, aż do osiągnięcia zakładanego celu
> **linker**
>
> inaczej konsolidator, program wchodzący w skład środowiska kompilatora; łączy on ze sobą skompilowane pliki i biblioteki statyczne, a także dodaje nagłówki (zawarta jest w nich informacja o maksymalnym rozmiarze stosu w systemie Windows); wynikiem działania linkera jest plik wykonywalny
> **nieskończona pętla**
>
> pętla, która nigdy nie spełni warunku kończącego jej działanie
> **rekurencja**
>
> sytuacja, w której funkcja wywołuje samą siebie, aż do napotkania przypadku podstawowego
> **rekursja**

inaczej: rekurencja

Prezentacja multimedialna

Problem 1

Używając obu metod (iteracyjnej i rekurencyjnej), napisz program w języku C++, który obliczy sumę kolejnych liczb naturalnych od 1 do n włącznie. Działanie algorytmu możesz wzorować na algorytmie obliczania silni.

Specyfikacja:

Dane:

- n – liczba naturalna dodatnia stanowiąca ostatni składnik sumy

Wynik:

- `suma` – liczba naturalna, suma liczb z przedziału $< 1, n >$

Polecenie 1

Zaimplementuj rozwiązanie metodą iteracyjną:

```cpp
#include <iostream>
using namespace std;

int main () {
    // Tutaj dodaj kod. Żeby coś wypisać, użyj polecenia: cout
}
```

```
1
```

Zaimplementuj rozwiązanie metodą rekurencyjną:

```cpp
#include <iostream>
using namespace std;

int main () {
    // Tutaj dodaj kod. Żeby coś wypisać, użyj polecenia: cout
}
```

```
1
```

Polecenie 2

Porównaj swoje rozwiązania z przedstawionymi w prezentacji.

1

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Rozpoczniemy implementację od metody iteracyjnej. Definiujemy funkcję o nazwie `sumaIteracyjna`, która będzie typu `int` i będzie miała jeden parametr. Parametr oznacza wartość, na której sumowanie się zakończy. W funkcji wykorzystamy zmienną `suma`, która będzie przechowywała dotychczasowy wynik, który z każdą iteracją będzie się powiększać.

```
1 int sumaIteracyjna(int n)
  {
2   int suma = 0;
3 }
```

Następnym krokiem będzie stworzenie pętli `for`.

2

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Potrzebujemy pętli `for`, która przeiteruje zakres liczb naturalnych od 1 do wartości n włącznie.

```
1 for (int i = 1; i <= n;
  i++) {
2   // Sumuj liczby
3 }
```

3

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Teraz wszystkie liczby, przez które iteruje nasza pętla `for`, musimy dodać do wcześniej już przygotowanej zmiennej `suma`.

```
for (int i = 1; i <= n;
i++) {
  suma += i;
}
```

Na koniec należy zwrócić wartość zmiennej `suma`:

```
return suma;
```

Nasze iteracyjne rozwiązanie jest już gotowe. Zobaczmy, jak wygląda gotowa funkcja `sumaIteracyjna()` i główna funkcja programu:

4

```
int sumaIteracyjna(int);

int main() {
  // Argument sumy
  int n = 5;

  sumaIteracyjna(n);

  return 0;
}

// Metoda iteracyjna
int sumaIteracyjna(int n)
{
  int suma = 0;

```

```
  for (int i = 1; i <= n; i++) {
      suma += i;
  }

  return suma;
}
```

5

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Zastanów się, jak będzie wyglądało rozwiązanie metodą rekurencyjną.

6

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Zacznijmy od deklaracji funkcji.

```
int sumaIteracyjna(int);
int sumaRekurencyjna(int);

int main() {
  // Argument sumy
  int n = 5;

  sumaIteracyjna(n);

  return 0;
}

// Metoda iteracyjna
int sumaIteracyjna(int n)
{
  int suma = 0;

```

```
  for (int i = 1; i <= n; i++) {
        suma += i;
  }

  return suma;
}

// Metoda rekurencyjna
int sumaRekurencyjna(int n) {

}
```

Umieściliśmy prototyp funkcji `sumaRekurencyjna` nad funkcją `main`, a definicję pod funkcją `main`.

Gdybyśmy pominęli deklarację prototypu nad funkcją `main`, kompilator w momencie przetwarzania funkcji `main` nie wiedziałby, że deklaracja tej funkcji nastąpi pod nią, co skutkowałoby błędem.

7

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Zgodnie z definicją, aby funkcję nazwać rekurencyjną musi ona wywoływać samą siebie.

Zadaniem przedstawionej poniżej funkcji `sumaRekurencyjna` jest zsumowanie kolejnych liczb od 1 do zadanego n.

Każde wywołanie naszej funkcji rekurencyjnej będzie dodawać otrzymany argument n do wartości zwracanej przez kolejne wywołanie funkcji. Ta częściowa suma będzie zwracana do funkcji wywołującej.

```
// Metoda rekurencyjna
```

```
int sumaRekurencyjna(int
n) {
    return n +
sumaRekurencyjna(n - 1);
}
```

8

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Pierwszy z dwóch niezbędnych elementów rekurencji już mamy. Gdybyśmy jednak uruchomili program w tej postaci, funkcja `sumaRekurencyjna` wywoływałaby się w nieskończoność. Dlatego potrzebujemy instrukcji warunkowej, która zwróci wartość 1, gdy argument n osiągnie wartość 1. Dodanie tego warunku jest niezbędne dla zakończenia rekurencji. Jest to zdefiniowanie tak zwanego przypadku bazowego lub przypadku końca wywołań rekurencyjnych.

```
// Metoda rekurencyjna
int sumaRekurencyjna(int
n) {
    if (n == 1) {
        return 1;
    }

    return n +
sumaRekurencyjna(n - 1);
}
```

9

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Mając zaimplementowane obie metody, możemy raz jeszcze spojrzeć na nasz program:

```
int sumaIteracyjna(int);
int sumaRekurencyjna(int);

int main() {
  // Argument sumy
  int n = 5;

  sumaIteracyjna(n);

  return 0;
}

// Metoda iteracyjna
int sumaIteracyjna(int n)
{
  int suma = 0;

  for (int i = 1; i <= n;
i++) {
      suma += i;
  }

  return suma;
}

// Metoda rekurencyjna
int sumaRekurencyjna(int
n) {
    if (n == 1) {
        return 1;
    }

    return n +
sumaRekurencyjna(n - 1);
}
```

Program powinien już działać – nie możemy jednak zobaczyć wyniku operacji. Wypiszmy więc wyniki do konsoli.

10

```cpp
#include <iostream>
using namespace std;

int sumaIteracyjna(int);
int sumaRekurencyjna(int);

int main() {
  // Argument sumy
  int n = 5;

  // Wypisanie wyników
  cout << "Suma iteracyjna: " << sumaIteracyjna(n) << endl;
  cout << "Suma rekurencyjna: " << sumaRekurencyjna(n) << endl;

  return 0;
}

// Metoda iteracyjna
int sumaIteracyjna(int n)
{
  int suma = 0;

  for (int i = 1; i <= n; i++) {
      suma += i;
  }

  return suma;
}

// Metoda rekurencyjna
int sumaRekurencyjna(int n) {
    if (n == 1) {
        return 1;
    }

    return n + sumaRekurencyjna(n - 1);
```

```
36 }
```

11

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/P11jZk8S9

Wczytajmy teraz liczbę n od użytkownika. Nasza specyfikacja wymaga, aby liczba n była większa od zera. Zweryfikujemy dane wejściowe i po podaniu danych zgodnych ze specyfikacją, obliczymy sumę. W przeciwnym razie, wypiszemy komunikat o błędnych danych.

```
1  #include <iostream>
2  using namespace std;
3
4  int sumaIteracyjna(int);
5  int sumaRekurencyjna(int);
6
7  int main() {
8    // Argument sumy
9    int n;
10   cin >> n;
11
12   if (n <= 0) {
13      cout << "Bledne dane.
   Podana liczba musi byc
   wieksza od zera." << endl;
14   } else {
15       // Wypisywanie
   wyników
16       cout << "Suma
   iteracyjna: " <<
   sumaIteracyjna(n) << endl;
17       cout << "Suma
   rekurencyjna: " <<
   sumaRekurencyjna(n) <<
   endl;
18   }
19
20   return 0;
```

```
}

// Metoda iteracyjna
int sumaIteracyjna(int n)
{
  int suma = 0;

  for (int i = 1; i <= n;
i++) {
      suma += i;
  }

  return suma;
}

// Metoda rekurencyjna
int sumaRekurencyjna(int
n) {
    if (n == 1) {
        return 1;
    }

    return n +
sumaRekurencyjna(n - 1);
}
```

Sprawdź się

Pokaż ćwiczenia: 

Ćwiczenie 1

Napisz program, który metodą iteracyjną wypisze w kolejności malejącej wszystkie liczby naturalne dodatnie nie większe od podanej liczby n i podzielne przez p. Załóż, że n i p są mniejsze od 10^6.

Przetestuj działanie programu dla n = 18 i p = 3.

Specyfikacja problemu:

Dane:

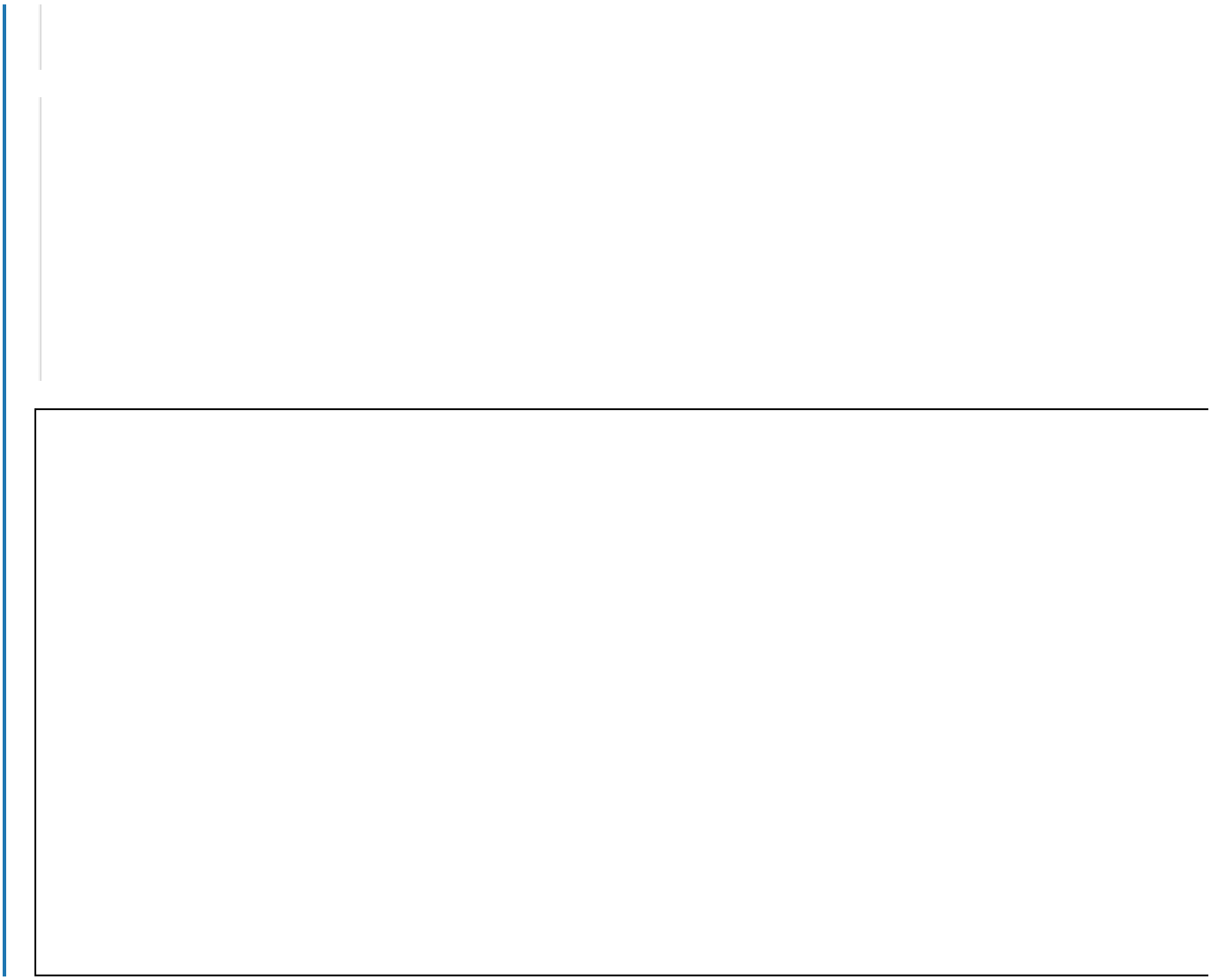
- n – liczba naturalna dodatnia, na której należy zakończyć sprawdzanie
- p – liczba naturalna dodatnia, która musi dzielić wypisywane liczby; p ≤ n

Wynik:

Program wypisuje wszystkie liczby naturalne dodatnie w kolejności malejącej podzielne przez p, które są nie większe niż podana liczba n.

Twoje zadania

1. Program wypisuje wszystkie liczby naturalne dodatnie w kolejności malejącej (każda w nowej linii) podzielne przez p, które są nie większe niż podana liczba n.



Ćwiczenie 2

Napisz program, który metodą rekurencyjną wypisze w kolejności malejącej wszystkie liczby naturalne dodatnie nie większe od podanej liczby n i podzielne przez p. Załóż, że n i p są mniejsze od 10^6.

Przetestuj swój program dla n = 18 i p = 2.

Specyfikacja problemu:

Dane:

- n – liczba naturalna dodatnia, na której należy zakończyć sprawdzanie
- p – liczba naturalna dodatnia, która musi dzielić wypisywane liczby; p ≤ n

Wynik:

Program wypisuje wszystkie liczby naturalne dodatnie w kolejności malejącej podzielne przez p, które są nie większe niż podana liczba n.

Twoje zadania

1. Program wypisuje wszystkie liczby naturalne dodatnie w kolejności malejącej podzielne przez p, które są nie większe niż podana liczba n. Każda liczba powinna być wyświetlona w nowej linii.

Ćwiczenie 3

Napisz program, który metodą iteracyjną wypisze sumę cyfr podanej liczby. W komentarzu znajduje się przykładowa implementacja wykonana metodą rekurencyjną. Przetestuj swój program dla liczby `473856394`.

Specyfikacja problemu:

Dane:

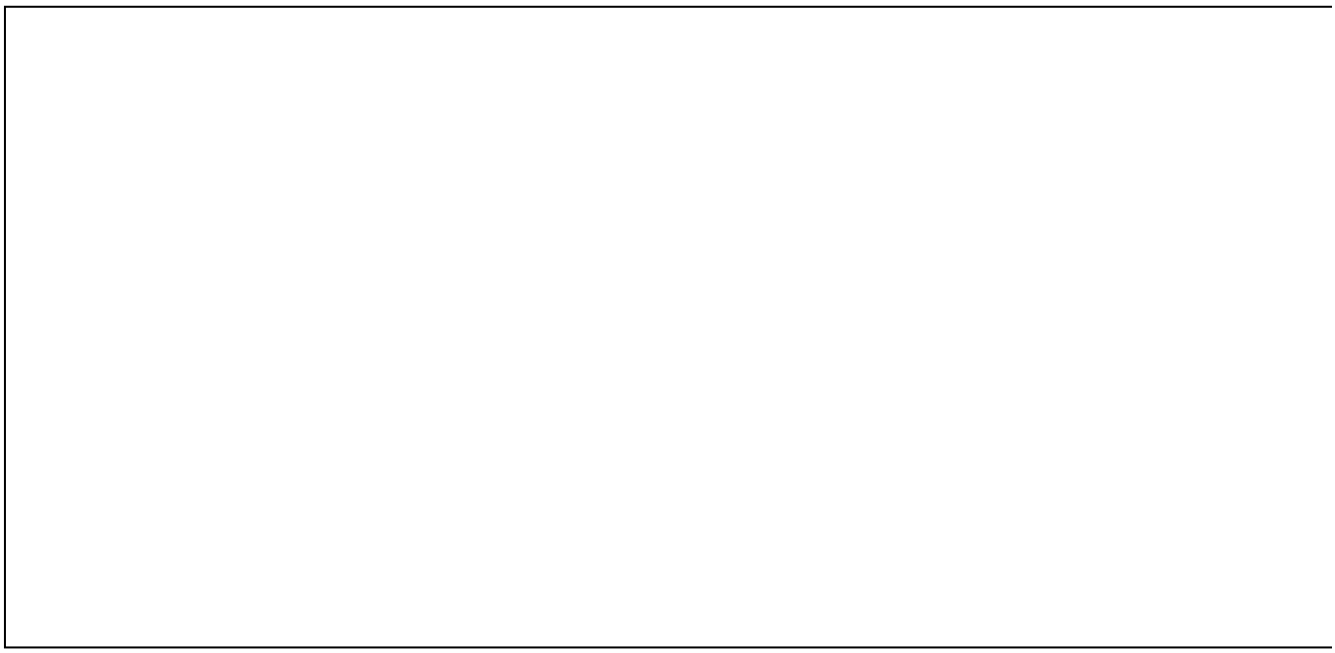
- n – liczba całkowita, której sumę cyfr należy obliczyć

Wynik:

- `suma` – liczba naturalna, suma cyfr liczby n

Twoje zadania

1. Program wypisuje sumę cyfr danej liczby.



Dla nauczyciela

Autor: Maurycy Gast
Przedmiot: Informatyka

Temat: Rekurencja a iteracja w języku C++

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

Treści nauczania – wymagania szczegółowe

I + II. Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) objaśnia, a także porównuje podstawowe metody i techniki algorytmiczne oraz struktury danych, wykorzystując przy tym przykłady problemów i algorytmów, w szczególności:

b) rekurencję (do generowania ciągów liczb, potęgowania, sortowania liczb, generowania fraktali),

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Podsumujesz informacje na temat podstawowych właściwości iteracji i rekurencji.

- Porównasz metodę iteracyjną i rekurencyjną w języku C++.
- Rozwiążesz zadany problem metodą rekurencyjną i iteracyjną.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Rekurencja a iteracja w języku C++". Nauczyciel prosi uczniów o zapoznanie się z treściami w sekcji „Przeczytaj".

Faza wstępna:

1. Wyświetlenie przez nauczyciela tematu i celów zajęć, przejście do wspólnego ustalenia kryteriów sukcesu.

Faza realizacyjna:

1. **Praca z tekstem.** Uczniowie przystępują do cichego czytania tekstu zawartego w sekcji „Przeczytaj", jeśli nauczyciel - na podstawie raportu na platformie - uważa, że przygotowanie uczniów jest wystarczające, może pominąć tę czynność.
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Prezentacja multimedialna". Uczniowie zapoznają się z problemem 1 i w parach piszą program w wersjach: iteracyjnej i rekurencyjnej. Następnie analizują prezentację i dyskutują

w parach, która z przedstawionych metod jest bardziej odpowiednia do rozwiązania tego problemu.
3. **Ćwiczenie umiejętności.** Prowadzący zapowiada uczniom, że będą rozwiązywać ćwiczenie nr 1 z sekcji „Sprawdź się". Każdy z uczniów robi to samodzielnie. Po ustalonym czasie następuje porównanie napisanych kodów podczas wspólnego omówienia rozwiązań.
4. Uczniowie w parach wykonują ćwiczenia nr 2 z sekcji „Sprawdź się". Nauczyciel sprawdza poprawność wykonanych zadań, omawiając je wraz z uczniami.

Faza podsumowująca:

1. Na koniec zajęć nauczyciel raz jeszcze wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie". W odniesieniu do ich realizacji dokonuje szczegółowej oceny rozwiązania zastosowanego przez wybranego ucznia.

Praca domowa:

1. Uczniowie wykonują ćwiczenie nr 3 z sekcji „Sprawdź się".

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka C++.
- Oficjalna dokumentacja techniczna dla kompilatora GCC/G++ 4.5 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Wskazówki metodyczne:

- Uczniowie mogą wykorzystać multimedium w sekcji „Prezentacja multimedialna" do przygotowania się do lekcji powtórkowej.