



Wykorzystanie rekurencji do rozwiązywania problemów algorytmicznych w C++

Veronika Tronchuk

I rok Informatyki

Nr albumu: 30019

Rekursja to metoda rozwiązywania problemów, w której funkcja lub algorytm wywołuje sam siebie w celu przetworzenia podproblemów. Pozwala to podzielić złożony problem na prostsze podproblemy tego samego typu.

1. Rekursywna definicja problemu:

Rekursja działa poprzez podział złożonego problemu na prostsze podproblemy tego samego rodzaju. Każde kolejne wywołanie rekursywne przetwarza mniejszą część problemu, aż do osiągnięcia przypadku podstawowego (najprostszego przypadku problemu).

2. Przypadek podstawowy:

Jest to warunek, który kończy rekursję. Bez niego rekursja byłaby nieskończona, co prowadziłoby do błędu. Przypadek podstawowy określa moment, w którym dalsze wywoływanie funkcji rekursywnej nie jest konieczne i można zwrócić wynik.

3. Przypadek rekursywny:

Jest to część funkcji, w której następuje jej ponowne wywołanie w celu rozwiązania podproblemu. Funkcja przetwarza część problemu i przekazuje jego uproszczoną wersję do kolejnego wywołania rekursywnego.

4. Głębokość rekursji:

Głębokość rekursji to liczba poziomów wywołań rekursywnych, które zostaną wykonane przed osiągnięciem przypadku podstawowego. W niektórych przypadkach może być ona bardzo duża.

5. Rekursja a iteracja:

Wiele algorytmów rekursywnych można zaimplementować iteracyjnie (np. przy użyciu pętli). Iteracja może być bardziej efektywna pod względem zużycia pamięci, ponieważ nie wymaga dodatkowego przechowywania kontekstu każdego wywołania funkcji.

6. Technika "dziel i zwyciężaj":

Rekursja jest podstawowym elementem technik takich jak **dziel i zwyciężaj** (divide and conquer). Problem jest dzielony na podproblemy, z których każdy jest rozwiązywany rekursywnie. Następnie wyniki podproblemów są łączone w celu uzyskania ostatecznego rozwiązania.

7. Rekursja a stos wywołań:

Każde wywołanie rekursywne jest przechowywane w **stosie wywołań**. Oznacza to, że system przechowuje kontekst wykonania każdej funkcji w pamięci, aby można było do niego wrócić po zakończeniu wykonania danej funkcji.

8. Memoizacja:

W niektórych przypadkach, gdy ten sam podproblem jest obliczany wielokrotnie, można zastosować **memoizację**. Jest to technika optymalizacji, polegająca na zapisywaniu wyników obliczeń dla podproblemów, aby uniknąć ich ponownego przetwarzania. Takie podejście jest stosowane np. w algorytmach programowania dynamicznego.

9. Generowanie i sprawdzanie rozwiązań:

Rekursja jest często wykorzystywana do generowania wszystkich możliwych wariantów rozwiązań (np. przeszukiwanie wszystkich ścieżek w grafie) lub do sprawdzania warunków (np. sprawdzanie poprawności rozwiązania zadania szachowego).

Kiedy stosować rekursję

- Gdy problem naturalnie dzieli się na mniejsze podproblemy o podobnej strukturze.
- Gdy struktura problemu pozwala wygodnie wyrazić rozwiązanie poprzez wywołanie rekursywne (np. w przypadku drzew, grafów, algorytmów wyszukiwania czy sortowania).
- Gdy problem ma dobrze określony przypadek podstawowy, który pozwala zatrzymać wywołania rekursywne.

Jak rozpoznać rekurencję w kodzie?

1. Funkcja wywołuje samą siebie

Główną cechą rekurencji jest to, że funkcja zawiera w swoim ciele wywołanie samej siebie. Może to być wywołanie bezpośrednie (funkcja wywołuje się samodzielnie) lub pośrednie (przez inne funkcje).

2. Obecność przypadku bazowego

Rekurencyjna funkcja powinna zawierać warunek, który określa moment zakończenia rekurencji. Przypadek bazowy zapobiega nieskończonemu wywoływaniu funkcji i pozwala na zwrócenie konkretnego wyniku.

3. Zmniejszanie rozmiaru problemu

Każde kolejne wywołanie rekurencyjne powinno operować na mniejszej lub uproszczonej wersji pierwotnego problemu. Dzięki temu program ostatecznie osiągnie przypadek bazowy i zakończy działanie.

4. Rekurencja może być wywoływana wielokrotnie w jednej funkcji

W niektórych przypadkach funkcja rekurencyjna może wywoływać samą siebie kilka razy w ramach jednego wywołania. Jest to typowe dla algorytmów operujących na strukturach drzewiastych.

5. Rekurencja jest często stosowana do pracy z rekurencyjnymi strukturami danych

Wiele algorytmów wykorzystuje rekurencję do operacji na takich strukturach jak drzewa, grafy czy do obliczeń matematycznych wymagających rekurencyjnego podejścia.

Zadanie 1: Obliczanie silni ($n!$) za pomocą rekurencji

Opis zadania:

Napisz funkcję rekurencyjną, która oblicza silnię liczby n , gdzie n jest liczbą całkowitą nieujemną. Funkcja powinna przyjmować jedną zmienną n , a następnie zwrócić wynik obliczenia $n!$.

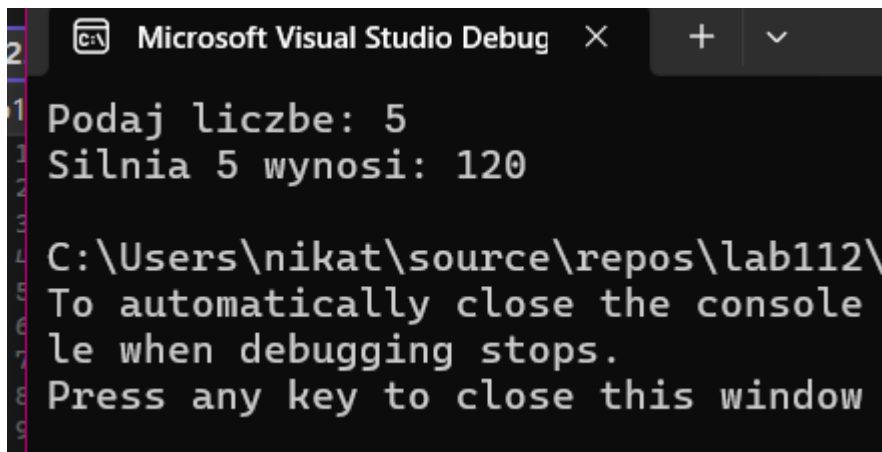
Kroki do wykonania:

1. Zaimplementuj funkcję rekurencyjną `silnia(int n)`, która zwróci wynik obliczenia silni.
2. Dodaj odpowiednią obsługę błędów (np. dla wartości ujemnych).
3. Zrób testy funkcji w `main()`, aby obliczyć silnię dla kilku liczb

```

1  #include <iostream>
2  using namespace std;
3
4  int silnia (int n){
5      if(n<=0){
6          return 1;
7      }
8      else{
9          return n * silnia (n-1);
10     }
11 }
12
13 int main(){
14     int n;
15     cout<<"Podaj liczbe: ";
16     cin>>n;
17     cout << "Silnia " << n << " wynosi: " << silnia(n) << endl;
18 }

```



```

2  Microsoft Visual Studio Debug  X  +  v
1  Podaj liczbe: 5
   Silnia 5 wynosi: 120
   C:\Users\nikat\source\repos\lab112\
   To automatically close the console
   le when debugging stops.
   Press any key to close this window

```

Analiza działania programu

Ten program implementuje rekurencyjne podejście do obliczania silni liczby. Składa się z dwóch głównych części: **funkcji obliczającej silnię** oraz **funkcji głównej**, która przetwarza dane wprowadzone przez użytkownika i wyświetla wynik.

1. Logika obliczania silni

Program wykorzystuje rekurencyjne podejście, w którym funkcja wywołuje samą siebie w celu obliczenia silni mniejszych liczb.

- **Przypadek bazowy**

Jeśli liczba jest mniejsza lub równa zero, funkcja zwraca jeden. Jest to konieczne, aby uniknąć nieskończonego wywoływania funkcji.

Dla silni istnieje matematyczna definicja: $0! = 1$. Program przestrzega tej zasady.

```
if (n <= 0) {  
    return 1;  
}
```

- **Wywołanie rekurencyjne**

Jeśli liczba jest większa od zera, program oblicza silnię jako iloczyn tej liczby i silni poprzedniej liczby.

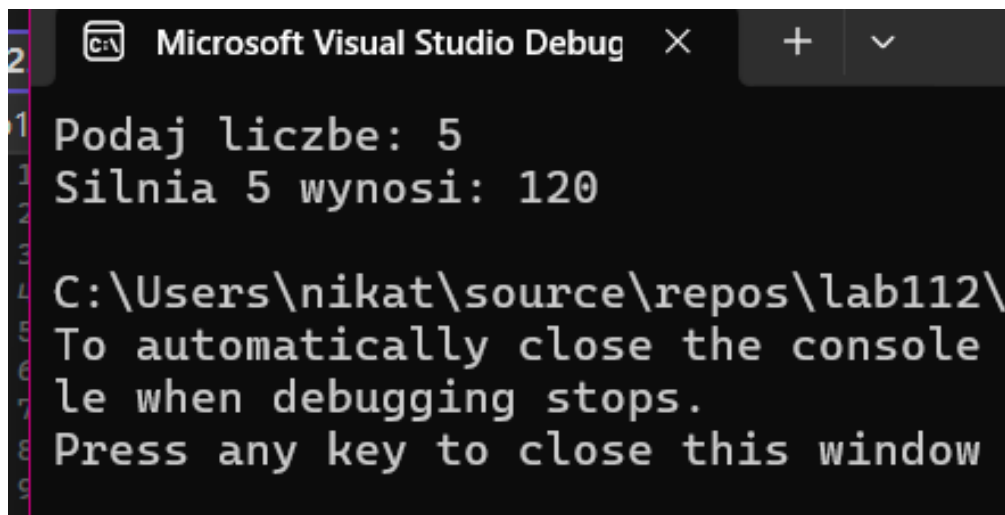
Tworzy to **łańcuch wywołań rekurencyjnych**, aż liczba nie zostanie zmniejszona do zera.

```
else {  
    return n * silnia(n - 1);  
}
```

2. Mechanizm działania rekurencji

Przyjrzyjmy się, co się dzieje, gdy użytkownik wprowadza liczbę 5:

1. Program sprawdza, czy 5 jest przypadkiem bazowym. Ponieważ tak nie jest, wywołuje funkcję dla liczby 4.
2. Następnie 4 także nie jest przypadkiem bazowym, więc wywoływana jest funkcja dla liczby 3.
3. Podobnie, program wywołuje funkcję dla liczby 2, potem dla 1, a na końcu dla 0.
4. Gdy program dochodzi do 0, zwraca 1 (przypadek bazowy).
5. Teraz wykonywane są wszystkie odkładane mnożenia w odwrotnej kolejności:
 - a. Silnia 2 -> $2 * 1 = 2$
 - b. Silnia 3 -> $2 * 3 = 6$
 - c. Silnia 4 -> $6 * 4 = 24$
 - d. Silnia 5 -> $24 * 5 = 120$
6. Ostateczny wynik (120) jest zwracany do funkcji głównej, która go wyświetla



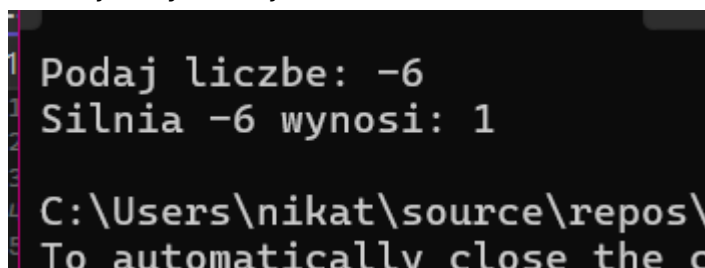
```
Microsoft Visual Studio Debug X + v
1 Podaj liczbe: 5
2 Silnia 5 wynosi: 120
3
4 C:\Users\nikat\source\repos\lab112\
5 To automatically close the console
6 when debugging stops.
7 Press any key to close this window
```

3. Funkcja główna: Interakcja z użytkownikiem

- Program wyświetla zapytanie o wprowadzenie liczby
- Użytkownik wprowadza liczbę, która jest przekazywana do funkcji obliczającej silnię
- Otrzymany wynik jest zwracany i wyświetlany na ekranie

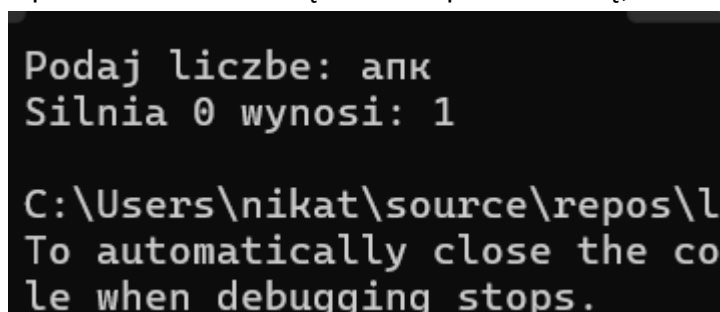
Możliwa wada:

- Jeśli użytkownik wprowadzi liczbę ujemną, funkcja zwróci 1, chociaż silnia dla liczb ujemnych nie jest zdefiniowana



```
Podaj liczbe: -6
Silnia -6 wynosi: 1
C:\Users\nikat\source\repos\
To automatically close the c
```

- Program nie sprawdza, czy wprowadzona wartość jest liczbą (jeśli użytkownik wpisze znaki lub liczbę zmiennoprzecinkową, może to prowadzić do błędów)



```
Podaj liczbe: anp
Silnia 0 wynosi: 1
C:\Users\nikat\source\repos\la
To automatically close the co
le when debugging stops.
```

Jak można ulepszyć program?

- Dodać sprawdzenie poprawności wprowadzonych danych
- Ograniczyć wprowadzanie tylko do liczb całkowitych nieujemnych
- Zastosować podejście iteracyjne zamiast rekurencyjnego, aby uniknąć głębokiej rekurencji i ewentualnego przepełnienia stosu przy dużych liczbach

Zadanie 2: Obliczanie n-tego elementu ciągu Fibonacciego

Opis zadania:

Ciąg Fibonacciego to ciąg liczb, w którym każda liczba jest sumą dwóch poprzednich. Pierwsze dwa elementy ciągu to 0 i 1, a kolejne elementy są obliczane zgodnie ze wzorem:

$$F(n)=F(n-1)+F(n-2)$$

Napisz funkcję rekurencyjną, która oblicza n-ty element ciągu Fibonacciego

Kroki do wykonania:

1. Zaimplementuj funkcję rekurencyjną `fibonacci(int n)`, która zwróci n-ty element ciągu Fibonacciego
2. Przetestuj funkcję w `main()`, obliczając pierwsze 10 elementów ciągu.
3. Wskaż, co się stanie, jeśli n będzie bardzo duże (np. > 40). Zwróć uwagę na czas wykonywania algorytmu


```

1  #include <iostream>
2  using namespace std;
3
4  int fibonacci(int n) {
5      if (n < 0) {
6          return 1;
7      }
8      else {
9          return fibonacci(n - 1) + fibonacci(n - 2);
10     }
11 }
12
13 int main() {
14     cout << "10 liczb Fibonacci: ";
15     for (int i = 0; i < 10; i++) {
16         cout << fibonacci(i) << " ";
17     }
18     cout << "\n";
19     int n;
20
21     cout << "Podaj liczbe: ";
22     cin >> n;
23     n -= 1;
24     cout << "n element Fibonacci " << n+1 << " wynosi: " << fibonacci(n) << endl;
25 }

```

```

10 liczb Fibonacci: 2 3 5 8 13 21 34 55 89 144
Podaj liczbe: 5
n element Fibonacci 5 wynosi: 13

```

C:\Users\nikat\source\repos\lab112\x64\Debug\lab11

Analiza działania programu

Ten program implementuje rekurencyjne podejście do obliczania liczb Fibonacciego. Składa się z dwóch głównych części:

1. **Funkcji obliczającej liczby Fibonacciego**, która wykorzystuje rekurencję.

```

    if (n < 0) {
        return 1;
    }
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

```

2. **Funkcji głównej**, która wyświetla pierwsze 10 liczb ciągu Fibonacciego oraz pozwala użytkownikowi wprowadzić liczbę, aby obliczyć jej wartość w tym ciągu

```
for (int i = 0; i < 10; i++) {  
    cout << fibonacci(i) << " ";  
}
```

1. Logika obliczania liczb Fibonacciego

Program wykorzystuje rekurencję do obliczania liczb Fibonacciego, gdzie każda liczba jest sumą dwóch poprzednich

2. Problemy w kodzie

- a. Algorytm działa bardzo wolno dla dużych wartości n

2. Wykładnicza złożoność

- a. Dla każdego n wywoływana jest funkcja dwukrotnie, co prowadzi do powtarzających się obliczeń.
- b. Na przykład, dla `fibonacci(5)` wywoływane są `fibonacci(4)` i `fibonacci(3)`, które z kolei wywołują `fibonacci(3)`, `fibonacci(2)` itd.

```
10 liczb Fibonacci: 2 3 5 8 13 21 34 55 89 144  
Podaj liczbę: 5  
n element Fibonacci 5 wynosi: 13
```

```
C:\Users\nikat\source\repos\lab112\x64\Debug\lab112.exe
```

- c. To sprawia, że algorytm jest bardzo wolny dla większych wartości n.

3. Wyświetlanie 10 liczb Fibonacciego

- a. Kod poprawnie wykonuje pętlę do wyświetlenia pierwszych 10 liczb.
- b. Jednak przez nieefektywną rekurencję może to zająć dużo czasu przy większych n.

```
10 liczb Fibonacci: 2 3 5 8 13 21 34 55 89 144  
Podaj liczbę: 40  
n element Fibonacci 40 wynosi: 267914296
```

```
C:\Users\nikat\source\repos\lab112\x64\Debug\lab112.exe
```

3. Jak ulepszyć program?

1. Sprawdzanie danych wejściowych

- Zabronić wprowadzania liczb ujemnych n.

```
10 liczb Fibonacci: 2 3 5 8 13 21 34 55 89 144
Podaj liczbę: -10
n element Fibonacci -10 wynosi: 1
C:\Users\nikat\source\repos\lab112\x64\Debug\lab112
```

- Obsługiwać dane wejściowe, które nie są liczbami (jeśli użytkownik wpisze znaki).

```
10 liczb Fibonacci: 2 3 5 8 13 21 34 55 89 144
Podaj liczbę: zap
n element Fibonacci 0 wynosi: 1
C:\Users\nikat\source\repos\lab112\x64\Debug\lab112
```

2. Poprawa wydajności

- Zastosować **programowanie dynamiczne**, aby uniknąć powtórnego obliczania.

Zadanie 3: Obliczanie sumy cyfr liczby

Opis zadania:

Napisz funkcję rekurencyjną, która oblicza sumę cyfr liczby całkowitej.

Kroki do wykonania:

- Zaimplementuj funkcję rekurencyjną `sumaCyfr(int n)`, która oblicza sumę cyfr liczby n.

2. Zauważ, że każdy krok polega na dodaniu ostatniej cyfry liczby i przejściu do pozostałej części liczby.
3. Testuj funkcję w main().

```
1  #include <iostream>
2  using namespace std;
3
4  int sumaCyfr(int n) {
5      if (n == 0)
6          return 0;
7
8      else
9          return n % 10 + sumaCyfr (n / 10);
10
11 }
12
13 int main() {
14     int n;
15     cout << "Podaj liczbe: ";
16     cin >> n;
17     cout << "Suma cyfr " << n << " wynosi: " << sumaCyfr(n) << endl;
18 }
```

```
Podaj liczbe: 123
Suma cyfr 123 wynosi: 6
```

Opis funkcji:

1. Funkcja sumaCyfr(int n):

- a. Jest to funkcja rekurencyjna, która oblicza sumę cyfr liczby.

```
int sumaCyfr(int n) {
    if (n == 0)
        return 0;

    else
        return n % 10 + sumaCyfr (n / 10);
}
```

- b. W przypadku bazowym (if (n == 0)) funkcja zwraca 0, co oznacza, że nie ma już żadnych cyfr do dodania.

```
if (n == 0)
    return 0;
```

- c. W przypadku rekurencyjnym (else), funkcja oblicza ostatnią cyfrę liczby za pomocą $n \% 10$ i dodaje ją do wyniku rekurencyjnego wywołania funkcji z liczbą, która jest o 1 cyfrę mniejsza ($n / 10$)

```
else
    return n % 10 + sumaCyfr (n / 10);
```

Przykład:

- d. Dla liczby 123:

- Pierwsze wywołanie: $123 \% 10 = 3$ (ostatnia cyfra) + wywołanie `sumaCyfr(12)`
- Drugie wywołanie: $12 \% 10 = 2$ (ostatnia cyfra) + wywołanie `sumaCyfr(1)`
- Trzecie wywołanie: $1 \% 10 = 1$ + wywołanie `sumaCyfr(0)`
- Przypadek bazowy: `sumaCyfr(0)` zwraca 0.
- Ostateczny wynik: $3 + 2 + 1 + 0 = 6$.

```
Podaj liczbe: 123
Suma cyfr 123 wynosi: 6
```

2. Funkcja `main()`:

```
int main() {
    int n;
    cout << "Podaj liczbe: ";
    cin >> n;
    cout << "Suma cyfr " << n << " wynosi: " << sumaCyfr(n) << endl;
}
```

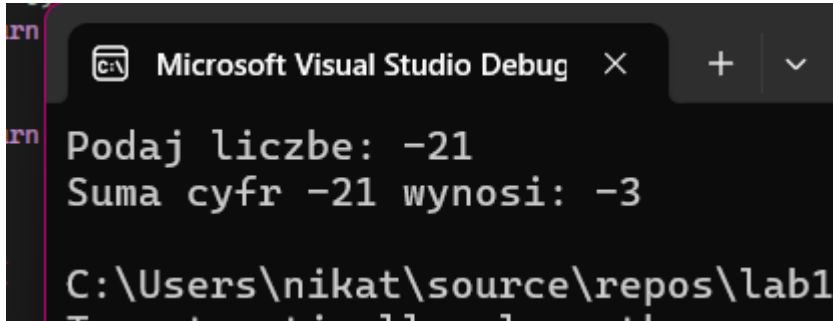
- Program prosi użytkownika o wprowadzenie liczby, którą zapisuje w zmiennej `n`.
- Następnie wywołuje funkcję `sumaCyfr(n)` i wyświetla wynik na ekranie.

Ogólny opis działania programu:

Program pozwala użytkownikowi na wprowadzenie liczby całkowitej. Następnie oblicza i wyświetla sumę cyfr tej liczby za pomocą funkcji rekurencyjnej `sumaCyfr`. Jest to doskonały przykład wykorzystania rekurencji do rozwiązania prostego zadania algorytmicznego.

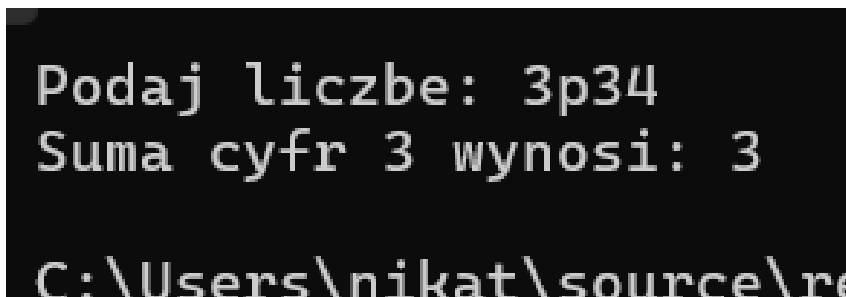
Możliwe ulepszenia:

1. **Sprawdzenie liczb ujemnych:** Ponieważ rekursja działa tylko z liczbami dodatnimi, warto dodać sprawdzenie, czy liczba nie jest ujemna. W przypadku liczby ujemnej, program mógłby wyświetlić odpowiedni komunikat.



```
Microsoft Visual Studio Debug X + v
Podaj liczbe: -21
Suma cyfr -21 wynosi: -3
C:\Users\nikat\source\repos\lab1
```

2. **Obsługa błędów wejścia:** Kod obecnie nie obsługuje sytuacji, w której użytkownik wprowadzi nieprawidłowe dane (np. tekst zamiast liczby). Można dodać sprawdzenie, które zapobiegne takim błędom.



```
Podaj liczbe: 3p34
Suma cyfr 3 wynosi: 3
C:\Users\nikat\source\re
```

Zadanie 4: Liczenie liczby wystąpień danej cyfry w liczbie

Opis zadania:

Napisz funkcję rekurencyjną, która liczy, ile razy dana cyfra występuje w liczbie całkowitej.

Kroki do wykonania:

1. Zaimplementuj funkcję rekurencyjną `liczbaWystapien(int n, int cyfra)`, która liczy wystąpienia cyfry `cyfra` w liczbie `n`.
2. Wykorzystaj operacje dzielenia i reszty z dzielenia do wyodrębnienia

poszczególnych cyfr.

3. Testuj funkcję w main().

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

int liczbaWystaoien(int n, int cyfra) {
    if (n == 0) {
        return 0;
    }
    else if (n % 10 == cyfra) {
        return 1 + liczbaWystaoien(n / 10, cyfra);
    }
    else {
        return liczbaWystaoien(n / 10, cyfra);
    }
}

int main() {
    int n, cyfra;
    cout << "Podaj liczbe: ";
    cin >> n;
    cout << "Podaj cyfre: ";
    cin >> cyfra;
    cout << "cyfra " << cyfra << " wystepuje: " << liczbaWystaoien(abs(n), cyfra) << endl;
}
```

```
i Podaj liczbe: 2223543
e Podaj cyfre: 2
{ cyfra 2 wystepuje: 3
e
C:\Users\nikat\source\repos\
```

Funkcja `liczbaWystaoien` jest funkcją rekurencyjną, która liczy, ile razy dana cyfra występuje w liczbie. Oto jak działa każda część kodu:

- Funkcja `liczbaWystaoien` przyjmuje dwa argumenty:
 - n: liczba, w której będziemy liczyć wystąpienia cyfry.
 - cyfra: cyfra, którą szukamy w liczbie.

```
int liczbaWystaoien(int n, int cyfra) {
```

- **Podejście rekurencyjne:**

Jeśli `n` stanie się równe 0, zwracamy 0 (przypadek bazowy, oznacza to, że nie ma już cyfr do sprawdzenia)

```
if (n == 0) {
    return 0;
}
```

Jeśli ostatnia cyfra liczby ($n \% 10$) jest równa poszukiwanej cyfrze, dodajemy 1 do wyniku i wywołujemy rekurencyjnie funkcję dla pozostałej części liczby.

```
else if (n % 10 == cyfra) {  
    return 1 + liczbaWystaoien(n / 10, cyfra);  
}
```

Jeśli ostatnia cyfra nie jest równa poszukiwanej cyfrze, po prostu wywołujemy funkcję dla liczby bez ostatniej cyfry.

```
else {  
    return liczbaWystaoien(n / 10, cyfra);  
}
```

Opis działania programu:

1. **Wejście:** Program pyta użytkownika o liczbę (n) i cyfrę ($cyfra$), której wystąpienia chcemy policzyć w tej liczbie.

```
int main() {  
    int n, cyfra;  
    cout << "Podaj liczbę: ";  
    cin >> n;  
    cout << "Podaj cyfrę: ";  
    cin >> cyfra;
```

2. **Obliczenia:** Funkcja `liczbaWystaoien` rekurencyjnie sprawdza wszystkie cyfry liczby i zlicza, ile razy poszukiwana cyfra występuje.

```
int liczbaWystaoien(int n, int cyfra) {  
    if (n == 0) {  
        return 0;  
    }  
    else if (n % 10 == cyfra) {  
        return 1 + liczbaWystaoien(n / 10, cyfra);  
    }  
    else {  
        return liczbaWystaoien(n / 10, cyfra);  
    }  
}
```

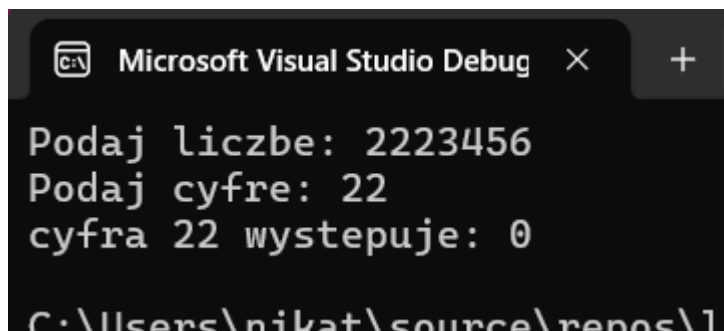
3. **Wyjście:** Program wyświetla liczbę wystąpień poszukiwanej cyfry w liczbie.

```
cout << "cyfra " << cyfra << " występuje: " << liczbaWystaoien(abs(n), cyfra) << endl;
```

Program nie sprawdza, czy użytkownik wprowadził poprawne dane. Na przykład, użytkownik może wprowadzić liczbę niecałkowitą lub liczbę zmiennoprzecinkową, co może prowadzić do nieprzewidywalnych wyników lub błędów.


```
Podaj liczbe: 45.34
Podaj cyfre: cyfra 0 wystepuje: 0
C:\Users\nikat\source\repos\lab112\x6
```

Program nie sprawdza również, czy wprowadzona cyfra jest faktycznie cyfrą (czyli liczbą od 0 do 9).



```
Microsoft Visual Studio Debug
Podaj liczbe: 2223456
Podaj cyfre: 22
cyfra 22 wystepuje: 0
C:\Users\nikat\source\repos\lab112\x6
```

Program używa typu `int`, który może nie wystarczyć do obsługi bardzo dużych liczb. Dla liczb mających więcej cyfr (np. 64-bitowych) może wystąpić przepełnienie zmiennej typu `int`.

Propozycje ulepszeń:

1. Sprawdzanie poprawności danych wejściowych:

Przed przetwarzaniem danych warto sprawdzić, czy użytkownik wprowadził poprawne liczby. Na przykład, warto sprawdzić, czy cyfra znajduje się w zakresie od 0 do 9.

Ogólne wnioski dotyczące rekurencji

Rekurencja jest potężnym narzędziem do rozwiązywania problemów, ale ma swoje ograniczenia, szczególnie gdy chodzi o wydajność i pracę z dużymi danymi. Ważne jest, aby rozumieć, że funkcje rekurencyjne mogą stać się nieefektywne, jeśli są wywoływane wielokrotnie dla tych samych wartości lub jeśli głębokość rekurencji staje się zbyt duża. W takich przypadkach warto rozważyć inne podejścia, takie jak iteracja,

które pozwalają uniknąć problemów z przepiętnieniem stosu i znacznie poprawiają wydajność programu.

Wnioski dotyczące zadania 1 (Obliczanie silni ($n!$) za pomocą rekurencji):

Program poprawnie realizuje rekurencyjne obliczanie silni, stosując logiczny podział na przypadek bazowy oraz wywołania rekurencyjne. Rekurencja pozwala na eleganckie rozwiązanie, jednak ma swoje ograniczenia, w szczególności znaczne zużycie pamięci przy dużych liczbach oraz możliwe błędy spowodowane brakiem sprawdzania danych wejściowych.

Aby zwiększyć efektywność, warto rozważyć alternatywne podejścia, takie jak iteracyjna metoda, optymalizacja rekurencji (np. rekurencja ogonowa), a także wdrożenie sprawdzania poprawności wprowadzonych danych oraz obsługi wyjątków.

Program działa poprawnie, ale można go ulepszyć w celu rozszerzenia funkcjonalności oraz poprawy wydajności.

Wnioski dotyczące zadania 2 (Obliczanie liczb Fibonacciego):

Program poprawnie realizuje rekurencyjne obliczanie liczb Fibonacciego, jednak jego wydajność jest nieefektywna dla dużych wartości n . Algorytm używa wywołań rekurencyjnych dla każdego elementu ciągu, co prowadzi do wielokrotnego obliczania tych samych wartości. To powoduje wykładniczy wzrost liczby operacji i znaczne spowolnienie pracy programu.

Rozwiązanie: Zastosowanie podejścia iteracyjnego, w którym liczby Fibonacciego obliczane są kolejno bez rekurencji, pozwoliłoby uniknąć głębokich wywołań rekurencyjnych i znacznie poprawić wydajność.

Wnioski dotyczące zadania 3 (Obliczanie sumy cyfr liczby):

1. **Sprawdzanie danych wejściowych w celu uniknięcia błędów:** Program powinien obsługiwać przypadki, w których użytkownik wprowadza niepoprawne

dane (np. litery lub znaki zamiast liczb), ponieważ może to prowadzić do błędów wykonania programu lub niepoprawnych wyników. Sprawdzanie danych wejściowych jest niezbędnym elementem każdej aplikacji, która przetwarza dane od użytkownika, aby uniknąć takich sytuacji.

2. **Poprawa doświadczeń użytkownika:** Wdrożenie sprawdzania danych wejściowych poprawia doświadczenia użytkownika, ponieważ program staje się bardziej niezawodny i wygodny w użyciu. Użytkownik nie otrzymuje nieoczekiwanych błędów, a zamiast tego dostaje jasne komunikaty o błędzie wprowadzonych danych i możliwość ich poprawienia.
3. **Zapobieganie błędom w trakcie wykonania:** Bez odpowiedniego sprawdzania danych wejściowych, program może zakończyć działanie lub wykonać niepoprawne operacje. Aby zapobiec takim sytuacjom, należy wcześniej zaplanować mechanizm obsługi wyjątków lub sprawdzania poprawności wprowadzonych danych.

Wnioski dotyczące zadania 4 (L Liczenie liczby wystąpień danej cyfry w liczbie):

1. **Brak sprawdzania poprawności wprowadzonych danych:** Program nie sprawdza poprawności danych wejściowych, co może prowadzić do nieprzewidywalnych wyników. Wprowadzenie sprawdzania danych (np. sprawdzenie, czy cyfra mieści się w zakresie od 0 do 9) pomogłoby uniknąć takich sytuacji. Ponadto, program nie obsługuje błędów, jeśli użytkownik wprowadzi niepoprawny format danych (np. liczby zmiennoprzecinkowe lub tekst). Dobrze byłoby dodać odpowiednią obsługę wyjątków dla takich przypadków.
2. **Rekurencyjne podejście i jego wydajność:** Rekurencyjne podejście, choć eleganckie, może prowadzić do problemów z wydajnością dla bardzo dużych liczb (np. liczb z milionami cyfr). Zbyt głęboka rekurencja może powodować przepełnienie stosu. W takim przypadku zastosowanie podejścia iteracyjnego mogłoby być bardziej efektywne.
3. **Wsparcie dla dużych liczb:** Program używa typu `int`, który nie nadaje się do obsługi bardzo dużych liczb. Dla liczb o dużej liczbie cyfr (np. 64-bitowych) może wystąpić przepełnienie zmiennej typu `int`. Rozważenie użycia typów danych z większą liczbą bitów (np. `long` lub nawet bibliotek do pracy z dużymi liczbami) pozwoliłoby uniknąć tego problemu.

Ogólne rekomendacje:

- Aby poprawić wydajność w programach rekurencyjnych, szczególnie przy pracy z dużymi danymi, warto przejść na algorytmy iteracyjne lub wykorzystać metody programowania dynamicznego, które pozwalają uniknąć powtarzających się obliczeń.
- Zawsze należy sprawdzać wprowadzone dane, aby zapewnić poprawne działanie programu i uniknąć błędów w wynikach.
- Wsparcie dla dużych liczb jest niezbędne dla bardziej elastycznej pracy z programami, które muszą obsługiwać wartości poza granicami standardowych typów danych.