

Materiały do przedmiotu:

Podstawy programowania

Laboratorium nr 5

Autor:

Patrycja Miazek



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	16
Pytania pomocnicze	22
Podsumowanie	22
Literatura	22



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Implementacja funkcji rekurencyjnych. Rozwiązywanie prostych problemów algorytmicznych

Wprowadzenie, tematyka zajęć

Podczas laboratorium studenci zapoznają się z implementacją funkcji rekurencyjnych w języku C#, które pozwalają rozwiązywać problemy poprzez wywoływanie tej samej funkcji w mniejszych, prostszych krokach. Ćwiczenia pokazują, jak rekurencja może zastąpić złożone pętle i struktury iteracyjne, umożliwiając czytelne rozwiązywanie problemów algorytmicznych. Studenci poznają mechanizm działania funkcji rekurencyjnych, znaczenie warunku brzegowego oraz fazy wchodzenia i powrotu wywołań, a także uczą się stosować rekurencję do klasycznych zadań matematycznych i algorytmicznych, takich jak obliczanie silni, wyznaczanie n-tego elementu ciągu Fibonacciego czy wyszukiwanie elementów w tablicach.

Cel ćwiczenia/laboratorium

Celem zajęć jest umożliwienie studentom praktycznego wykorzystania funkcji rekurencyjnych do rozwiązywania prostych problemów algorytmicznych w języku C#. Studenci uczą się, jak definiować funkcje rekurencyjne, jak dobierać odpowiedni warunek brzegowy oraz jak analizować przepływ wywołań funkcji. Zajęcia pozwalają zrozumieć mechanizm działania rekurencji, stosować ją do obliczeń matematycznych i przetwarzania danych oraz tworzyć programy, które w sposób przejrzysty i efektywny rozwiązują zadania wymagające wielokrotnego powtarzania operacji. Dzięki ćwiczeniom studenci rozwijają umiejętności projektowania algorytmów rekurencyjnych i przygotowują się do bardziej złożonych zagadnień w programowaniu.

1. Instrukcja laboratoryjna/ćwiczeniowa

Każda **funkcja rekurencyjna** składa się z dwóch istotnych elementów. Pierwszym z nich jest **warunek brzegowy**, który określa moment **zakończenia rekurencji**. To sytuacja, w której funkcja **nie wywołuje już samej siebie i zwraca wynik bez dalszych odwołań**. Drugim elementem jest **wywołanie rekurencyjne**, czyli **odwołanie funkcji do samej siebie** z prostszym argumentem, przybliżającym rozwiązanie do warunku brzegowego. Każde wywołanie trafia na **stos wywołań**, gdzie przechowywany jest



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



kontekst bieżącego wywołania. Po osiągnięciu warunku brzegowego funkcje „zwijają się” po stosie, zwracając wyniki kolejno do wcześniejszych wywołań. Dla przykładu funkcja obliczająca silnię liczby n w C# może wyglądać tak:

```
int Silnia(int n)
{
    if (n == 0)
        return 1;
    else
        return n * Silnia(n - 1);
}
```

Mechanizm działania polega na tym, że każde kolejne wywołanie przekazuje mniejszą wartość parametru, aż osiągnięty zostanie warunek prosty ($n == 0$). Następnie funkcje zaczynają zwracać wyniki w odwrotnej kolejności, budując ostateczny rezultat.

Rodzaje rekurencji

Rekurencję można podzielić na kilka typów. **Rekurencja prosta**, zwana też bezpośrednią, oznacza, że funkcja wywołuje samą siebie. **Rekurencja pośrednia** pojawia się wtedy, gdy jedna funkcja wywołuje inną, a ta z kolei wywołuje funkcję pierwotną. **Rekurencja liniowa** to taka, w której funkcja wywołuje samą siebie tylko raz w każdym przebiegu, natomiast **rekurencja drzewiasta**, jak w przypadku ciągu Fibonacciego, może wywoływać funkcję wielokrotnie w jednym kroku. Istnieje także **rekurencja ogonowa**, w której wywołanie rekurencyjne jest ostatnią instrukcją funkcji. Pozwala to na optymalizację stosu wywołań przez kompilator.

Przebieg wykonania rekurencji

Działanie rekurencji można podzielić na dwie fazy. Pierwsza to **faza wchodzenia**, podczas której funkcje **wywołują same siebie**, upraszczając problem aż do osiągnięcia warunku brzegowego. Druga to **faza powrotu**, w której wyniki zaczynają być **zwracane w odwrotnej kolejności** do wcześniejszych wywołań. Dzięki temu rekurencja działa podobnie jak pętla, ale jest często bardziej naturalna i elegancka przy problemach o strukturze drzewiastej lub hierarchicznej.

Zastosowania rekurencji



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Rekurencja jest szeroko stosowana w informatyce. W matematyce używa się jej do wyznaczania silni, największego wspólnego dzielnika czy kolejnych liczb Fibonacciego. W strukturach danych pozwala na przeszukiwanie drzew i grafów, a także ułatwia operacje na tablicach i napisach, takie jak sumowanie elementów, wyszukiwanie czy odwracanie ciągów znaków. Algorytmy sortowania, takie jak sortowanie szybkie czy przez scalanie, również naturalnie wykorzystują rekurencję. Dodatkowo przydaje się przy generowaniu kombinacji, permutacji i wariacji, analizie drzew składniowych, wyrażeń logicznych czy przetwarzaniu katalogów i plików, gdzie każdy folder może zawierać podfoldery, tworząc naturalną strukturę rekurencyjną.

Zalety i wady rekurencji

Zaletą rekurencji jest jej **prostota i czytelność**. Kod rekurencyjny jest często krótszy i bardziej naturalny niż iteracyjny, szczególnie przy zadaniach hierarchicznych. Ułatwia zapis matematycznych zależności i pozwala uniknąć dodatkowych zmiennych pomocniczych oraz złożonych pętli.

Wadą rekurencji jest **większe zużycie pamięci**, ponieważ każde wywołanie odkłada dane na stosie. Przy dużej liczbie wywołań może dojść do **spadku wydajności** lub **błędu przepełnienia stosu**. W C# rekurencja ogonowa nie jest automatycznie optymalizowana, więc przy głębokich wywołaniach warto rozważyć iteracyjne rozwiązania lub stosowanie struktur danych symulujących stos. Brak poprawnego warunku brzegowego może prowadzić do nieskończonego wywoływania funkcji.

Rekurencja a iteracja

Chociaż rekurencja i iteracja często prowadzą do tych samych wyników, różnią się mechanizmem działania. **Iteracja opiera się na powtarzaniu kroków w pętli**, natomiast **rekurencja polega na wielokrotnym wywoływaniu tej samej funkcji**. Każdy problem możliwy do rozwiązania rekurencyjnie można również zapisać iteracyjnie, choć często kosztem czytelności. Przykładem jest obliczanie silni, które można równie dobrze zrealizować za pomocą pętli for.

Optymalizacja rekurencji

W C# rekurencja nie jest automatycznie optymalizowana, co oznacza, że przy głębokich wywołaniach może dojść do błędu przepełnienia stosu. W takich



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



przypadkach można zastąpić rekurencję iteracją, użyć struktur danych symulujących stos lub spróbować zastosować rekurencję ogonową, jeśli problem na to pozwala.

Przykład 5.1

Napisz program, który oblicza silnię liczby n .

Przykładowe rozwiązanie:

```
Console.Write("Podaj n: ");
int n = int.Parse(Console.ReadLine());

int Silnia(int x)
{
    if (x == 0)
    {
        return 1; // przypadek bazowy
    }
    else
    {
        return x * Silnia(x - 1); // rekurencja
    }
}

Console.WriteLine(n + "! = " + Silnia(n));
```

Modyfikacje do samodzielnego testowania:

2. Zmień typ zmiennej na long, aby obliczać większe silnie (np. $20!$).
3. Sprawdź działanie programu dla liczby 0 i 1 – zobaczysz, że $Silnia(0) = 1$, $Silnia(1) = 1$.

Przykład 5.2

Napisz program, który sumuje wszystkie liczby od 1 do n rekurencyjnie.

Przykładowe rozwiązanie:

```
Console.Write("Podaj liczbę n: ");
int n = int.Parse(Console.ReadLine());

int Suma(int x)
{

```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
    if (x == 0) return 0; // warunek brzegowy
    return x + Suma(x - 1); // wywołanie rekurencyjne
}

Console.WriteLine("Suma liczb od 1 do " + n + " wynosi: " +
Suma(n));
```

Modyfikacje do samodzielnego testowania:

1. Zmieniaj funkcję, aby sumować tylko liczby parzyste.
2. Sprawdź, co się stanie, gdy podasz liczbę 0 lub 1.

Przykład 5.3

Napisz program, który znajduje n-tą liczbę w ciągu Fibonacciego.

Przykładowe rozwiązanie:

```
Console.Write("Podaj numer liczby w ciągu Fibonacciego: ");
int n = int.Parse(Console.ReadLine());
int Fibonaccini(int x)
{
    if (x == 0) return 0; // warunek brzegowy
    if (x == 1) return 1; // warunek brzegowy
    return Fibonaccini(x - 1) + Fibonaccini(x - 2); // wywołanie
    rekurencyjne
}
Console.WriteLine("Liczba Fibonacciego o numerze " + n + " wynosi: "
+ Fibonaccini(n));
```

Modyfikacje do samodzielnego testowania:

1. Dodaj opcję wyświetlenia wszystkich liczb od 0 do n.
2. Sprawdź, co się dzieje dla małych i dużych wartości n (np. n = 30).

Przykład 5.4

Napisz program, który rekurencyjnie odwraca ciąg znaków.

Przykładowe rozwiązanie:

```
Console.Write("Podaj tekst do odwrócenia: ");
string tekst = Console.ReadLine();
string Odwroc(string s)
{
    if (s.Length <= 1) return s; // warunek brzegowy
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
        return Odwroc(s.Substring(1)) + s[0]; // wywołanie rekurencyjne  
    }  
    Console.WriteLine("Odwrócony tekst: " + Odwroc(tekst));
```

Modyfikacje do samodzielnego testowania:

1. Spróbuj odwrócić zdanie z spacjami.
2. Zmieniaj funkcję, aby ignorowała wielkość liter.

Przykład 5.5

Napisz program, który rekurencyjnie sprawdza, czy liczba jest palindromem.

Przykładowe rozwiązanie:

```
Console.Write("Podaj liczbę: ");  
int liczba = int.Parse(Console.ReadLine());  
  
bool CzyPalindrom(int x, int odwrocona = 0)  
{  
    if (x == 0) return true;  
    int ostatniaCyfra = x % 10;  
    int reszta = x / 10;  
    return CzyPalindrom(reszta, odwrocona * 10 + ostatniaCyfra);  
}  
Console.WriteLine("Liczba " + liczba + (CzyPalindrom(liczba) ? "  
jest palindromem." : " nie jest palindromem
```

Modyfikacje do samodzielnego testowania:

1. Sprawdź działanie dla liczb jednocyfrowych.
2. Zmodyfikuj funkcję, aby działała z liczbami ujemnymi.

Przykład 5.6

Napisz program, który rekurencyjnie oblicza sumę cyfr liczby.

Przykładowe rozwiązanie:

```
Console.Write("Podaj liczbę: ");  
int liczba = int.Parse(Console.ReadLine());  
  
int SumaCyfr(int x)
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską




```
{  
    if (x == 0) return 0; // warunek brzegowy  
    return (x % 10) + SumaCyfr(x / 10); // wywołanie rekurencyjne  
}  
Console.WriteLine("Suma cyfr liczby: " + SumaCyfr(liczba));
```

Modyfikacje do samodzielnego testowania:

1. Sprawdź działanie funkcji dla liczby 0.
2. Zmieniaj liczby i sprawdzaj, czy wynik się zgadza.

Przykład 5.7

Napisz program, który rekurencyjnie liczy długość tekstu.

Przykładowe rozwiązanie:

```
Console.Write("Podaj tekst: ");  
string tekst = Console.ReadLine();  
  
int Dlugosc(string s)  
{  
    if (s == "") return 0; // warunek brzegowy  
    return 1 + Dlugosc(s.Substring(1)); // wywołanie rekurencyjne  
}  
Console.WriteLine("Długość tekstu: " + Dlugosc(tekst));
```

Modyfikacje do samodzielnego testowania:

1. Spróbuj policzyć długość pustego tekstu.
2. Testuj teksty z spacjami i znakami specjalnymi.

Przykład 5.8

Napisz program, który oblicza NWD dwóch liczb rekurencyjnie.

Przykładowe rozwiązanie:

```
Console.Write("Podaj pierwszą liczbę: ");  
int a = int.Parse(Console.ReadLine());  
Console.Write("Podaj drugą liczbę: ");  
int b = int.Parse(Console.ReadLine());  
int NWD(int x, int y)  
{  
    if (y == 0) return x; // warunek brzegowy  
    return NWD(y, x % y); // wywołanie rekurencyjne  
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
}  
Console.WriteLine("NWD liczb " + a + " i " + b + " wynosi: " +  
NWD(a, b));
```

Modyfikacje do samodzielnego testowania:

1. Sprawdź, co się stanie, gdy jedna z liczb to 0.
2. Przetestuj działanie dla dwóch takich samych liczb.

Przykład 5.9

Napisz program, który rekurencyjnie sprawdza, czy tekst zawiera tylko cyfry.

Przykładowe rozwiązanie:

```
Console.Write("Podaj tekst: ");  
string tekst = Console.ReadLine();  
bool CzySameCyfry(string s)  
{  
    if (s.Length == 0) return true; // warunek brzegowy  
    if (!char.IsDigit(s[0])) return false;  
    return CzySameCyfry(s.Substring(1));  
}  
Console.WriteLine("Tekst " + (CzySameCyfry(tekst) ? "zawiera tylko  
cyfry." : "zawiera inne znaki."));
```

Modyfikacje do samodzielnego testowania:

1. Sprawdź działanie dla pustego ciągu.
2. Dodaj warunek, aby tekst mógł zawierać także spację.

Przykład 5.10

Napisz program, który rekurencyjnie rysuje linię gwiazdek w konsoli.

Przykładowe rozwiązanie:

```
void Gwiazdki(int n)  
{  
    if (n == 0) return; // warunek brzegowy  
    Console.Write("*");  
    Gwiazdki(n - 1); // wywołanie rekurencyjne  
}  
Console.Write("Podaj liczbę gwiazdek: ");
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
int n = int.Parse(Console.ReadLine());  
Gwiazdki(n);
```

Modyfikacje do samodzielnego testowania:

1. Zmień funkcję tak, aby rysowała gwiazdki w kilku liniach.
2. Dodaj możliwość wprowadzenia znaku innego niż *.

Przykład 5.11

Napisz program, który rekurencyjnie sprawdza, czy w tablicy znajduje się określony element.

Przykładowe rozwiązanie:

```
bool Szukaj(int[] tab, int i, int szukana)  
{  
    if (i >= tab.Length) return false; // warunek brzegowy  
    if (tab[i] == szukana) return true;  
    return Szukaj(tab, i + 1, szukana); // wywołanie rekurencyjne  
}  
int[] liczby = { 3, 8, 1, 5, 9 };  
Console.Write("Podaj liczbę do wyszukania: ");  
int x = int.Parse(Console.ReadLine());  
Console.WriteLine(Szukaj(liczby, 0, x) ? "Znaleziono!" : "Nie  
znaleziono.");
```

Modyfikacje do samodzielnego testowania:

1. Zmień funkcję tak, aby zwracała indeks znalezionego elementu.
2. Dodaj obsługę pustej tablicy.

Przykład 5.12

Napisz program, który oblicza sumę liczb parzystych od 1 do n.

Przykładowe rozwiązanie:

```
int SumaParzystych(int n)  
{  
    if (n == 0) return 0; // warunek brzegowy  
    if (n % 2 == 0)  
        return n + SumaParzystych(n - 1);  
    else  
        return SumaParzystych(n - 1);  
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
}  
  
Console.Write("Podaj liczbę n: ");  
int n = int.Parse(Console.ReadLine());  
Console.WriteLine("Suma liczb parzystych od 1 do " + n + " wynosi: "  
+ SumaParzystych(n));
```

Modyfikacje do samodzielnego testowania:

1. Zmień funkcję tak, aby sumowała liczby nieparzyste.
2. Dodaj możliwość podania zakresu od m do n.

Przykład 5.13

Napisz program, który oblicza n-ty wyraz ciągu geometrycznego.

Przykładowe rozwiązanie:

```
double CiagGeometryczny(double a1, double q, int n)  
{  
    if (n == 1) return a1; // warunek brzegowy  
    return q * CiagGeometryczny(a1, q, n - 1); // wywołanie  
    rekurencyjne  
}  
  
Console.Write("Podaj pierwszy wyraz ciągu: ");  
double a1 = double.Parse(Console.ReadLine());  
Console.Write("Podaj iloraz q: ");  
double q = double.Parse(Console.ReadLine());  
Console.Write("Podaj numer wyrazu n: ");  
int n = int.Parse(Console.ReadLine());  
Console.WriteLine($"{n}-ty wyraz ciągu geometrycznego wynosi:  
{CiagGeometryczny(a1, q, n)}");
```

Modyfikacje do samodzielnego testowania:

1. Sprawdź działanie dla $q < 1$ i $q > 1$.
2. Zmień funkcję, aby obliczała sumę n pierwszych elementów ciągu.

Przykład 5.14

Napisz program, który rekurencyjnie znajduje najmniejszy element tablicy..

Przykładowe rozwiązanie:

```
int Minimum(int[] tab, int i)
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
{  
    if (i == tab.Length - 1) return tab[i]; // warunek brzegowy  
    int min = Minimum(tab, i + 1); // rekurencja  
    return (tab[i] < min) ? tab[i] : min;  
}  
int[] dane = { 12, 5, 7, 3, 9, 2 };  
Console.WriteLine("Najmniejszy element tablicy to: " + Minimum(dane,  
0));
```

Modyfikacje do samodzielnego testowania:

1. Dodaj wersję funkcji, która zwraca również największy element.
2. Przetestuj działanie dla tablicy z jednym elementem.

Przykład 5.15

Napisz program, który rekurencyjnie zamienia liczbę dziesiętną na binarną.

Przykładowe rozwiązanie:

```
string NaBinarny(int n)  
{  
    if (n == 0) return "0";  
    if (n == 1) return "1";  
    return NaBinarny(n / 2) + (n % 2).ToString(); // rekurencja  
}  
Console.Write("Podaj liczbę: ");  
int liczba = int.Parse(Console.ReadLine());  
Console.WriteLine("Liczba binarna: " + NaBinarny(liczba));
```

Modyfikacje do samodzielnego testowania:

1. Sprawdź działanie dla liczb 0, 1, 5, 16, 255.
2. Dodaj wersję funkcji, która zamienia liczbę binarną na dziesiętną.

Przykład 5.16

Napisz program, który rekurencyjnie oblicza potęgę liczby całkowitej..

Przykładowe rozwiązanie:

```
Console.Write("Podaj podstawę: ");  
int a = int.Parse(Console.ReadLine());  
Console.Write("Podaj wykładnik: ");  
int n = int.Parse(Console.ReadLine());
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
int Potega(int x, int y)
{
    if (y == 0) return 1;
    return x * Potega(x, y - 1);
}

Console.WriteLine($"{a}^{n} = {Potega(a, n)}");
```

Modyfikacje do samodzielnego testowania:

1. Spróbuj obsłużyć wykładnik ujemny (zwracając wartość typu double).
2. Zmodyfikuj funkcję, aby wykorzystywała optymalizację przez dzielenie wykładnika na połowy.

Przykład 5.17

Napisz program, który rekurencyjnie sprawdza, czy liczba jest pierwsza.

Przykładowe rozwiązanie:

```
Console.Write("Podaj liczbę: ");
int n = int.Parse(Console.ReadLine());

bool CzyPierwsza(int x, int dzielnik = 2)
{
    if (x < 2) return false;
    if (dzielnik * dzielnik > x) return true;
    if (x % dzielnik == 0) return false;
    return CzyPierwsza(x, dzielnik + 1);
}

Console.WriteLine(CzyPierwsza(n) ? "Liczba jest pierwsza" : "Liczba nie jest pierwsza");
```

Modyfikacje do samodzielnego testowania:

1. Spróbuj sprawdzić liczby bardzo duże, np. $n = 9973$.
2. Dodaj opcję wypisania wszystkich dzielników sprawdzanych podczas rekurencji.

Przykład 5.18

Napisz program, który rekurencyjnie odwraca cyfrę liczby całkowitej.

Przykładowe rozwiązanie:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
Console.Write("Podaj liczbę: ");
int n = int.Parse(Console.ReadLine());

int Odwroc(int x, int odwrocona = 0)
{
    if (x == 0) return odwrocona;
    return Odwroc(x / 10, odwrocona * 10 + x % 10);
}

Console.WriteLine("Odwrócona liczba: " + Odwroc(n));
```

Modyfikacje do samodzielnego testowania:

1. Spróbuj liczby jednocyfrowe oraz 0.
2. Zmień funkcję, aby działała również dla liczb ujemnych.

Przykład 5.19

Napisz program, który rekurencyjnie sumuje cyfry parzyste liczby.

Przykładowe rozwiązanie:

```
Console.Write("Podaj liczbę: ");
int n = int.Parse(Console.ReadLine());

int SumaParzystych(int x)
{
    if (x == 0) return 0;
    int cyfra = x % 10;
    return (cyfra % 2 == 0 ? cyfra : 0) + SumaParzystych(x / 10);
}

Console.WriteLine("Suma cyfr parzystych: " + SumaParzystych(n));
```

Modyfikacje do samodzielnego testowania:

1. Zmień funkcję, aby sumowała cyfry nieparzyste.
2. Sprawdź działanie dla liczby 0 i bardzo dużych liczb.

Przykład 5.20

Napisz program, który rekurencyjnie znajduje największą cyfrę liczby.

Przykładowe rozwiązanie:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
Console.Write("Podaj liczbę: ");  
int n = int.Parse(Console.ReadLine());  
  
int NajwiekszaCyfra(int x)  
{  
    if (x < 10) return x;  
    int reszta = NajwiekszaCyfra(x / 10);  
    return (x % 10 > reszta) ? x % 10 : reszta;  
}  
  
Console.WriteLine("Największa cyfra: " + NajwiekszaCyfra(n));
```

Modyfikacje do samodzielnego testowania:

1. Zmień funkcję, aby znalazła również najmniejszą cyfrę.
2. Sprawdź działanie dla liczby jednocyfrowej i wielocyfrowej.

Przykład 5.21

Napisz program, który rekurencyjnie sumuje cyfry liczby binarnej.

Przykładowe rozwiązanie:

```
Console.Write("Podaj liczbę binarną: ");  
string bin = Console.ReadLine();  
  
int SumaBin(string b)  
{  
    if (b.Length == 0) return 0;  
    return (b[0] - '0') + SumaBin(b.Substring(1));  
}  
  
Console.WriteLine("Suma cyfr binarnych: " + SumaBin(bin));
```

Modyfikacje do samodzielnego testowania:

1. Sprawdź działanie dla pustego ciągu i liczby "0".
2. Dodaj funkcję, która konwertuje liczbę binarną na dziesiętną rekurencyjnie.

Ćwiczenia samodzielne

Zadanie 5.1

1. Wczytaj od użytkownika tablicę 10 liczb całkowitych.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



2. Rekurencyjnie policz, ile w tablicy jest liczb dodatnich, a ile ujemnych. W każdej iteracji sprawdzaj ostatni element tablicy i wywołuj funkcję dla reszty.

Zadanie 5.2

1. Poproś użytkownika o podanie liczby n .
2. Narysuj w konsoli trójkąt z gwiazdek o wysokości n , używając rekurencji. Najpierw narysuj krótsze linie, a dopiero potem dłuższą.

Zadanie 5.3

1. Poproś użytkownika o podanie liczby całkowitej.
2. Sprawdź rekurencyjnie, czy liczba ta jest liczbą pierwszą.

Zadanie 5.4

1. Poproś użytkownika o podanie słowa.
2. Rekurencyjnie sprawdź, ile w tym słowie jest samogłosek.

Zadanie 5.5

1. Wczytaj od użytkownika dowolny tekst.
2. Napisz rekurencyjną funkcję, która zamienia wszystkie małe litery na duże. W każdym kroku zamień pierwszy znak na wielką literę i dodaj resztę z wywołania rekurencyjnego.

Zadanie 5.6

1. Poproś użytkownika o podanie liczby całkowitej n .
2. Wypisz rekurencyjnie wszystkie liczby od 1 do n , które są podzielne przez 3.

Zadanie 5.7

1. Poproś użytkownika o podanie tablicy 6 liczb.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



2. Napisz program, który rekurencyjnie policzy średnią arytmetyczną tych liczb. Najpierw policz sumę elementów, a potem podziel ją przez długość tablicy.

Zadanie 5.9

1. Napisz rekurencyjną funkcję, która odwraca tablicę liczb w miejscu (zamienia element pierwszy z ostatnim, drugi z przedostatnim itd.).

Zadanie 5.10

1. Napisz rekurencyjną wersję algorytmu sortowania bąbelkowego (bubble sort).
2. Program powinien działać na tablicy podanej przez użytkownika. Jedno wywołanie rekurencyjne przesuną największy element na koniec.

Zadanie 5.11

1. Poproś użytkownika o podanie liczby całkowitej a i wykładnika n .
2. Napisz funkcję rekurencyjną, która oblicza a^n .

Podpowiedź: $a^n = a * a^{n-1}, a^0 = 1$.

Zadanie 5.12

1. Poproś użytkownika o podanie liczby całkowitej n .
2. Napisz funkcję rekurencyjną, która obliczy silnię liczby n zgodnie z zasadą: $n! = n \cdot (n-1)!$ dla $n > 0$, a $0! = 1$.
3. Wyświetl wynik w konsoli.

Zadanie 5.13

1. Poproś użytkownika o podanie liczby całkowitej.
2. Napisz funkcję rekurencyjną, która obliczy sumę cyfr liczby, wykorzystując mechanizm: $\text{suma}[n] = n \% 10 + \text{suma}[n/10]$.
3. Wyświetl wynik w konsoli.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 5.14

1. Poproś użytkownika o wpisanie słowa lub zdania.
2. Napisz funkcję rekurencyjną, która sprawdzi, czy tekst jest palindromem. Porównuj pierwszy i ostatni znak, a następnie wywołaj funkcję dla środkowej części tekstu.
3. Wyświetl wynik w konsoli.

Zadanie 5.16

1. Poproś użytkownika o podanie tablicy 10 liczb.
2. Napisz funkcję rekurencyjną, która znajdzie największy element w tablicy.
Podpowiedź: Porównuj ostatni element z wynikiem rekurencji dla reszty tablicy.

Zadanie 5.17

1. Poproś użytkownika o podanie liczby n .
2. Napisz funkcję rekurencyjną, która wypisze pierwsze n liczb Fibonacciego.
Podpowiedź: $F(n)=F(n-1)+F(n-2)$, $F(0)=0$, $F(1)=1$.

Zadanie 5.18

1. Poproś użytkownika o podanie tablicy liczb.
2. Rekurencyjnie sprawdź, czy tablicę można podzielić na dwie części o równej sumie.
Podpowiedź: Spróbuj dodać lub nie dodać elementu do jednej grupy i sprawdź rekurencyjnie pozostałe elementy.

Zadanie 5.19

1. Poproś użytkownika o podanie liczby krążków n .
2. Zaimplementuj program który rekurencyjnie rozwiąże problem Wieży Hanoi, polegający na przenoszeniu krążków z pręta A na pręt C, z wykorzystaniem



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



pręta pomocniczego B. Program powinien wyświetlać kolejne kroki przenoszenia krążków w odpowiedniej kolejności.

3. Zastosuj funkcję rekurencyjną, w której:

- najpierw przenosisz $n-1$ krążków z A na B,
- następnie przenosisz 1 krążek z A na C,
- a na końcu przenosisz $n-1$ krążków z B na C.

4. Podpowiedź: Funkcja rekurencyjna powinna przyjmować cztery argumenty: liczbę krążków oraz nazwy trzech prętów (źródłowego, pomocniczego i docelowego).

Zadanie 5.20

1. Poproś użytkownika o podanie tablicy liczb.
2. Napisz funkcję rekurencyjną, która wypisze wszystkie możliwe podzbiory tablicy.

Zadanie 5.21

1. Poproś użytkownika o wpisanie tekstu.
2. Napisz funkcję rekurencyjną, która zliczy ile razy w tekście pojawia się podana litera.

Zadanie 5.22

1. Poproś użytkownika o dwie liczby całkowite a i b.
 2. Napisz funkcję rekurencyjną, która oblicza ich iloczyn bez użycia operatora *.
- Podpowiedź: $a \cdot b = a + a \cdot (b-1)$.

Zadanie 5.23

1. Poproś użytkownika o dwie liczby całkowite a i b ($b \neq 0$).
2. Napisz funkcję rekurencyjną, która obliczy wynik dzielenia całkowitego a / b bez użycia operatora /. Odejmuj b od a aż do momentu, gdy $a < b$.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 5.24

1. Poproś użytkownika o wpisanie zdania.
2. Napisz funkcję rekurencyjną, która zliczy ile słów jest w zdaniu.

Zadanie 5.25

1. Poproś użytkownika o podanie liczby całkowitej.
2. Napisz funkcję rekurencyjną, która sprawdzi, czy liczba jest parzysta.

Zadanie 5.26

1. Poproś użytkownika o podanie liczby całkowitej.
2. Rekurencyjnie wypisz wszystkie cyfry liczby od lewej do prawej.

Zadanie 5.27

1. Poproś użytkownika o podanie tablicy liczb.
2. Napisz funkcję rekurencyjną, która zsumuje wszystkie możliwe ciągłe podtablice tablicy. Dodawaj lub pomijaj każdy element w podtablicy i wywołuj funkcję dla reszty.

Zadanie 5.28

1. Poproś użytkownika o podanie dwóch słów.
2. Napisz funkcję rekurencyjną, która sprawdzi, czy jedno słowo jest anagramem drugiego (czyli czy po przestawieniu liter jednego słowa można utworzyć drugie słowo).

Zadanie 5.29

1. Poproś użytkownika o podanie planszy labiryntu (2D lista: 0 – ściana, 1 – droga).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



2. Napisz funkcję rekurencyjną, która sprawdzi, czy można przejść z lewego górnego rogu do prawego dolnego. Rekurencyjnie próbuj ruchu w czterech kierunkach, unikając powrotów.

Zadanie 5.30

1. Poproś użytkownika o podanie liczby n .
2. Napisz funkcję rekurencyjną, która obliczy liczbę wszystkich permutacji n -elementowego zbioru $\{n!\}$ ($n! = n \cdot (n-1)!$).

Pytania pomocnicze

1. Jak wytłumaczyłbyś własnymi słowami, czym jest rekurencja?
2. Co to jest warunek brzegowy (warunek stopu) i dlaczego jest konieczny w każdej funkcji rekurencyjnej?
3. Co się stanie, jeśli w funkcji rekurencyjnej zabraknie warunku brzegowego?
4. Na czym polega różnica między rozwiązaniem problemu za pomocą pętli a rozwiązaniem rekurencyjnym?
5. Podaj przykład problemu, który łatwo rozwiązać rekurencją, a trudniej pętlą.

Podsumowanie

Podczas laboratorium studenci zdobyli praktyczne doświadczenie w implementacji funkcji rekurencyjnych w języku C#. Ćwiczenia pozwoliły zrozumieć, jak definiować funkcje wywołujące same siebie, jak stosować warunki brzegowe oraz jak analizować przepływ wywołań w fazie wchodzenia i powrotu. Studenci nauczyli się rozwiązywać klasyczne problemy algorytmiczne, takie jak obliczanie silni, elementów ciągu Fibonacciego czy wyszukiwanie wartości w strukturach danych. Dzięki temu zdobyli umiejętności tworzenia przejrzystych i efektywnych programów rekurencyjnych, co stanowi solidną podstawę do dalszej nauki algorytmiki i bardziej złożonych zagadnień programowania w języku C#.

Literatura

Burns, Sean (2019). Hands-On Network Programming with C# and .NET Core: Build Robust Network Applications with C# and .NET Core. Wydanie 1. Birmingham: Packt Publishing, Limited.



Fundusze Europejskie
dla Rozwoju Społecznego



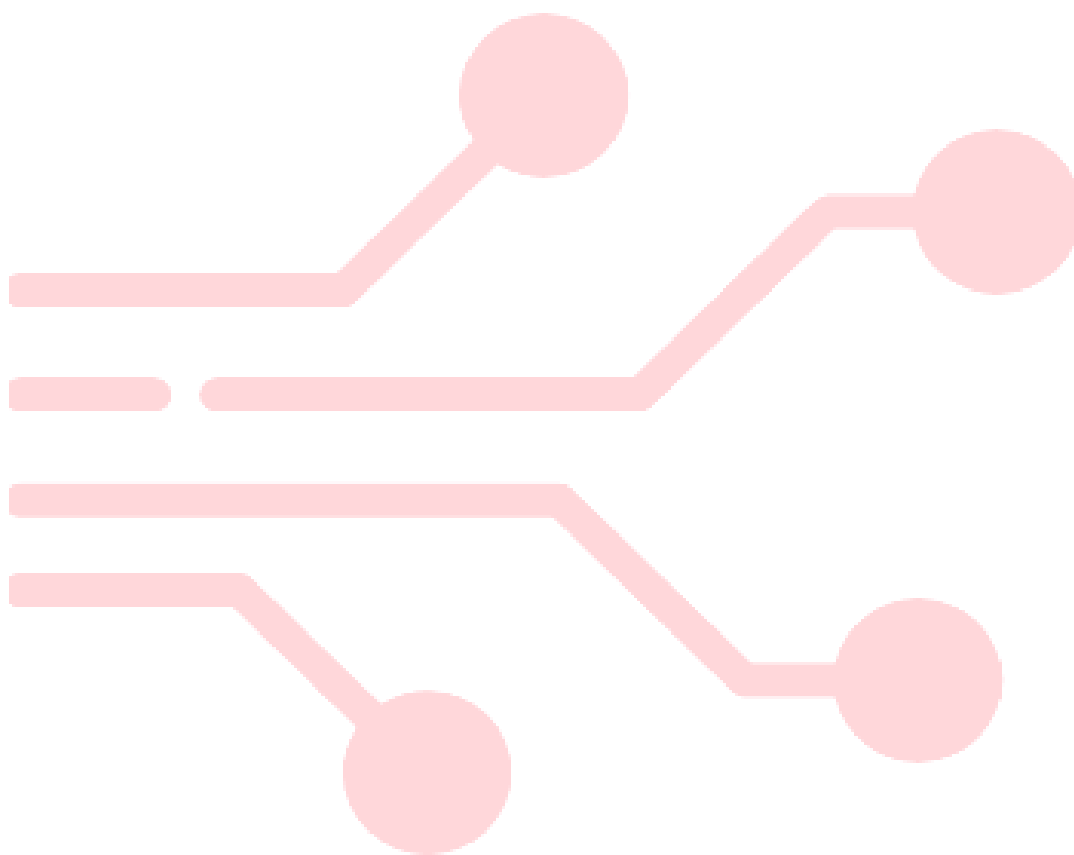
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Michaelis, Mark; Lippert, Eric [red.]; Walczak, Tomasz [tłum.]; Torgersen, Mads [przedm.] [2020]. C# 7.0 : kompletny przewodnik dla praktyków. Gliwice: Helion.

Alls, Jason; Meryk, Radosław [tłum.] [2021]. Czysty kod w C# : techniki refaktoryzacji i najlepsze praktyki. Gliwice: Helion.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

