# Spis treści

Spis treści

[Spis treści 1](#_Toc102216812)

[[T2] Algorytmy, złożoność obliczeniowa, pseudokod, struktury danych (kolejki, stosy), iteracja, rekurencja, algorytm Euklidesa. 2](#_Toc102216813)

[1. Algorytm 2](#_Toc102216814)

[2. Pseudokod 2](#_Toc102216815)

[3. Złożoność czasowa 2](#_Toc102216816)

[4. Kolejka 2](#_Toc102216817)

[5. Stos 3](#_Toc102216818)

[6. Algorytm iteracyjny 3](#_Toc102216819)

[7. Algorytm rekurencyjny 3](#_Toc102216820)

[8. Algorytm Euklidesa 3](#_Toc102216821)

[9. Test z zagadnienia: 4](#_Toc102216822)

[[T3] Sortowanie przez zliczanie, scalanie, ograniczenie dolne na złożoność algorytmów sortowania 4](#_Toc102216823)

[1. O sortowaniu 4](#_Toc102216824)

[2. Pseudokod sortowania przez scalanie 4](#_Toc102216825)

[3. Pseudokod sortowania przez zliczanie 5](#_Toc102216826)

[4. Analiza algorytmu sortowania przez zliczanie 6](#_Toc102216827)

[5. Test z zagadnień 6](#_Toc102216828)

[[T4] Grafy (proste), multigrafy, drzewa, algorytmy grafowe: DFS 7](#_Toc102216829)

[[T5] Algorytmy grafowe: BFS 7](#_Toc102216830)

[[T6] Kopce, sortowanie przez kopcowanie 7](#_Toc102216831)

[[T7] Najkrótsze drogi w grafach, algorytm Dijkstry, Bellmana-Forda 7](#_Toc102216832)

[[T8] Najkrótsze drzewa rozpinające, algorytm Prima, Kruskala 7](#_Toc102216833)

[[T9] Drzewa przeszukiwań binarnych 7](#_Toc102216834)

[[T10] Grafy Eulera, problem chińskiego listonosza 7](#_Toc102216835)

[[T11 – T12] Grafy Hamiltona, problem komiwojażera, algorytmy przeszukiwania z nawrotami i aproksymacyjne, problemy NP-trudne, NP-zupełne. 7](#_Toc102216836)

[[T13] Algorytmy tekstowe, kodowanie Huffmana 7](#_Toc102216837)

# [T2] Algorytmy, złożoność obliczeniowa, pseudokod, struktury danych (kolejki, stosy), iteracja, rekurencja, algorytm Euklidesa.

## Algorytm

Algorytm to dobrze zdefiniowany ciąg jednoznacznych instrukcji przeprowadzających dane wejściowe w dane wyjściowe. Przykłady algorytmów to:

* algorytm Euklidesa
* algorytmy sortowania
* algorytmy wyszukiwania.

Algorytmy służą do przedstawienia instrukcji prowadzących do rozwiązania zadanego problemu. Najpopularniejsze sposoby zapisu algorytmu to:

* lista kroków
* pseudokod
* zapis w konkretnym języku programowania
* opis słowny
* schemat blokowy

## Pseudokod

Pseudokod to jeden ze sposobów zapisu algorytmu. Zapis przypomina implementację w języku programowania, jednak z usunięciem wszystkich instrukcji charakterystycznych dla konkretnego języka programowania (np. deklaracji zmiennych). Bloki kodu oznaczamy graficznie wcięciami (tabulacje, spacje). Przykłady pseudokodów znajdują się w rozwiązaniu zadania 7. Pseudokody stosuje się powszechnie do zapisu algorytmów.

## Złożoność czasowa

Złożoność obliczeniowa (czasowa) to liczba operacji dominujących wykonanych w algorytmie. Operacja dominująca jest podana i jest to najczęściej: operacja arytmetyczna, logiczna, przypisanie, porównanie.  
Notacja dużego [O](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=O): [f](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=f)  jest co najwyżej rzędu [g](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=g), co oznaczamy jako [f(n)\in O(g(n))](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=f%28n%29%5Cin%20O%28g%28n%29%29) jeśli dla pewnego [n_0 > 0](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=n_0%20%3E%200) istnieje stała [c>0](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=c%3E0) taka, że dla wszystkich [n>n_0](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=n%3En_0) zachodzi nierówność [f(n) \leq c\cdot g(n)](https://plas.mat.umk.pl/moodle/filter/tex/displaytex.php?texexp=f%28n%29%20%5Cleq%20c%5Ccdot%20g%28n%29).

## Kolejka

Kolejka to struktura danych przechowująca dowolną liczbę elementów. Nowe elementy możemy dodawać wyłącznie na koniec kolejki, a zdejmować elementy możemy tylko z początku kolejki zgodnie z zasadą: *First In, First Out* (FIFO) *-* pierwszy na wejściu, pierwszy na wyjściu. Podstawowe operacje na kolejce to:

* **enqueue** - dodanie elementu na koniec kolejki
* **dequeue** - pobranie elementu z początku kolejki

## Stos

Stos to struktura danych przechowująca dowolną liczbę elementów. Nowe elementy możemy dodawać wyłącznie na koniec stosu, zdejmować elementy możemy też tylko z końca stosu zgodnie z zasadą: *Last In, First Out* (LIFO) - ostatni na wejściu, pierwszy na wyjściu. Podstawowe operacje na stosie to:

* **push** - dodanie elementu na koniec stosu
* **pop** - pobranie elementu z końca stosu

## Algorytm iteracyjny

Algorytm iteracyjny (funkcja iteracyjna) wykonuje w pętli tą samą operację (z góry określoną liczbę razy lub do spełnienia wskazanego warunku). Przykładowy algorytm iteracyjny podany jest w rozwiązaniu zadania 7, gdzie operacja odejmowania powtarzana jest do chwili, gdy wartości zmiennych a oraz b będą równe.

## Algorytm rekurencyjny

Algorytm rekurencyjny (funkcja rekurencyjna) wywołuje sam siebie. Ważnym elementem algorytmów rekurencyjnych jest warunek stopu, czyli informacja przy spełnieniu jakich warunków algorytm zwraca konkretną wartość zamiast wywołać siebie. Przykładowy algorytm rekurencyjny podany jest w rozwiązaniu zadania 7, gdzie warunkiem stopu jest osiągnięcie przez zmienną b wartości zero (w takiej sytuacji zwracana jest w algorytmie wartość zmiennej a).

## Algorytm Euklidesa

Algorytm Euklidesa wyznacza największy wspólny dzielnik dwóch liczb naturalnych, tzn. największą liczbę naturalną, która dzieli bez reszty obie z podanych liczb.

Wersja iteracyjna algorytmu z odejmowaniem:

function NWD(a, b)

while a != b

if a > b  
 a = a - b

else

b = b - a   
 return a

Wersja rekurencyjna algorytmu z operacją modulo:

function NWD(a, b)

if b = 0

return a

else

return NWD(b, a mod b)

## Test z zagadnienia:

1. Pseudokod to:
   1. Jeden ze sposobów zapisu algorytmu.
   2. Nie jest sposobem zapisu algorytmu.
2. Podana definicja „*Elementy mogą być wstawiane […] w dowolnej kolejności ale usunąć można tylko ten element, który został najwcześniej wstawiony”* opisuje:
   1. Stos
   2. Kolejkę

# [T3] Sortowanie przez zliczanie, scalanie, ograniczenie dolne na złożoność algorytmów sortowania

## O sortowaniu

Załóżmy, że chcemy posortować *n* różnych elementów *x*1,…,*xn* i jedynym sposobem ustalenia porządku pomiędzy nimi jest porównywanie ich w parach. Wynikiem sortowania jest permutacja *xi*1<*xi2*<…<*x*in. Możliwych wyników sortowania jest oczywiście tyle, ile **wszystkich permutacji** zbioru *n*-elementowego, czyli **n!**. Każdy algorytm sortowania przez porównania można zapisać za pomocą **drzewa decyzyjnego**.

**Drzewo decyzyjne** jest drzewem **binarnym** w którym wszystkie węzły różne od liści mają po dwóch synów. W tych węzłach zapisujemy pary indeksów *i*:*j*, 1≤*i*<*j*≤*n*. Każda taka para *i*:*j* oznacza, że interesuje nas wynik porównania *xi* z *xj*.

Niech *S*(*n*) oznacza minimalną liczbę porównań zawsze wystarczającą do posortowania *n* elementów. Liczbę *S*(*n*)można z dołu oszacować za pomocą drzewa decyzyjnego.

Pesymistyczna złożoność algorytmu opisanego za pomocą drzewa decyzyjnego, to oczywiście **wysokość** tego drzewa, czyli długość najdłuższej ścieżki od korzenia do liścia w tym drzewie mierzona liczbą krawędzi. Drzewo decyzyjne sortujące *n* elementów jest drzewem binarnym o **n!** liściach. Najmniejsza wysokość drzewa binarnego o *k*  liściach wynosi ⌈log*k*⌉.   Wynika stąd, że każdy algorytm sortujący przez porównania wykonuje w pesymistycznym przypadku *C*(*n*) = ⌈log*n*!⌉         porównań. Można pokazać, że ⌈log*n*!⌉≥*n*log*n*−1.45*n*

Zatem zachodzi*S*(*n*) ≥ *C*(*n*) = ⌈log*n*!⌉ ≥ *n*log*n*−1.45*n*

## Pseudokod sortowania przez scalanie

// A - tablica/lista do posortowania wielkości n

function MergeSort(A, n):

**if** n > 1:

m = n/2

A1 = A[0, m, m - 1]

A2 = A[m, n-m, n - 1]

MergeSort(?......)

MergeSort(?......)

Scal(A1, A2, A, m, n - m + 1)

// L1, L2 - scalane w L3 tablice/listy wielkości odpowiednio n1 i n2

function Scal(L1, L2, L3, n1, n2):

i1 = 0; i2 = 0; i3 = 0

**while** i1 < n1 **and** i2 < n2:

**if** L1[i1] < L2[i2]:

L3[i3] = L1[i1]

i1 = i1 + 1

**else**:

L3[i3] = L2[i2]

i2 = i2 + 1

i3 = i3 + 1

**while** i1 < n1:

L3[i3] = L1[i1]

i1 = i1 + 1

i3 = i3 + 1

**while** i2 < n2:

L3[i3] = L2[i2]

i2 = i2 + 1

i3 = i3 + 1

## Pseudokod sortowania przez zliczanie

funkcja zlicz(tablica, ogr\_min, ogr\_max):

    // można założyć, że ogr\_min = 0

    inicjujemy tablicę liczba\_wystapien rozmiaru ogr\_max - ogr\_min + 1

    wypełniamy tablicę liczba\_wystapien zerami

    inicjujemy tablicę wynikowa roziaru takiego jak tablica

    dla każdej wartości element w tablicy tablica:

        liczba\_wystapien[element] = liczba\_wystapien[element] + 1

    dla indeksu indeks w przedziale [ogr\_min, ogr\_max]:

        dla i w przedziale [0, liczba\_wystapien[indeks]]:

            do tablicy wynikowa wpisz wartość indeks

    zwróć wynikowa

## Analiza algorytmu sortowania przez zliczanie

Start algorytmu

1. Dla danych tablica = [1, 3, 0, 1, 0, 1, 1, 3]  
   mamy: gr\_min = 0, ogr\_max = 3
2. liczba\_wystapien = [0, 0, 0, 0]
3. wynikowa = [0, 0, 0, 0, 0, 0, 0, 0]
4. liczba\_wystapien = [2, 4, 0, 2]
5. **wynikowa = [0, 0, 1, 1, 1, 1, 3, 3]**

Koniec działania algorytmu

## Test z zagadnień

1. Pesymistyczna złożoność algorytmu opisanego za pomocą drzewa decyzyjnego to oczywiście [\_\_\_\_\_\_\_\_\_\_\_\_] tego drzewa, czyli długość najdłuższej ścieżki od korzenia do liścia w tym drzewie mierzona liczbą krawędzi
   1. Wysokość
   2. Długość
2. W pierwszym etapie sortowania przez zliczanie otrzymano następującą tabelę pomocniczą [1, 2, 3, 4, 5] o indeksach 0 – 4. Wynikowa posortowana na jej podstawie tablica to:
   1. [1, 2, 3, 4, 5]
   2. [0, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4]

# [T4] Grafy (proste), multigrafy, drzewa, algorytmy grafowe: DFS

# [T5] Algorytmy grafowe: BFS

# [T6] Kopce, sortowanie przez kopcowanie

# [T7] Najkrótsze drogi w grafach, algorytm Dijkstry, Bellmana-Forda

# [T8] Najkrótsze drzewa rozpinające, algorytm Prima, Kruskala

# [T9] Drzewa przeszukiwań binarnych

# [T10] Grafy Eulera, problem chińskiego listonosza

# [T11 – T12] Grafy Hamiltona, problem komiwojażera, algorytmy przeszukiwania z nawrotami i aproksymacyjne, problemy NP-trudne, NP-zupełne.

# [T13] Algorytmy tekstowe, kodowanie Huffmana