Algorytmy i Struktury Danych - Szczegółowe Notatki

Spis Treści

Wprowadzenie

Algorytmy i Struktury Danych są podstawą informatyki, umożliwiając efektywne rozwiązywanie problemów poprzez optymalny dobór i implementację struktur danych oraz algorytmów. Wiedza ta jest kluczowa dla tworzenia wydajnych aplikacji, optymalizacji procesów oraz rozwoju nowych technologii.

Kluczowe Cele Notatek:

- Zrozumienie różnych abstrakcyjnych struktur danych i ich implementacji.
- Nauka podstawowych algorytmów oraz technik algorytmicznych.
- Umiejętność analizowania i konstruowania algorytmów.
- Praktyczne zastosowanie struktur danych i algorytmów w programowaniu.

Abstrakcyjne Struktury Danych

Abstrakcyjne struktury danych definiują logiczny sposób organizowania i przechowywania danych, umożliwiając efektywne wykonywanie operacji na tych danych. W zależności od potrzeb aplikacji, różne struktury danych oferują różne zalety i wady.

Listy

Opis: Lista to sekwencja elementów ułożonych w określonym porządku, gdzie każdy element jest identyfikowany przez swoje miejsce w sekwencji.

Rodzaje List:

1. **Lista Jednokierunkowa (Singly Linked List)**
2. **Lista Dwukierunkowa (Doubly Linked List)**
3. **Lista Tablicowa (ArrayList)**

Lista Jednokierunkowa

Opis: Struktura danych, w której każdy element (węzeł) zawiera wartość oraz wskaźnik do następnego elementu w liście.

Implementacja:

```
class Node:
    def __init__(self, data):
        self.data = data  # Przechowywana wartość
        self.next = None  # Wskaźnik na następny węzeł

class SinglyLinkedList:
    def __init__(self):
        self.head = None  # Początek listy

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node  # Jeśli lista jest pusta, nowy węzeł jest głową
            return
        last = self.head
        while last.next:
            last = last.next  # Przejdź do końca listy
        last.next = new_node  # Dodaj nowy węzeł na końcu

    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        print(" -> ".join(map(str, elements)))
```

Przykład Użycia:

```
ll = SinglyLinkedList()
ll.append(10)
ll.append(20)
ll.append(30)
ll.display()  # Wyjście: 10 -> 20 -> 30
```

Złożoność Operacji:

- **Dodawanie na końcu:** O(n)
- **Dodawanie na początku:** O(1)
- **Wyszukiwanie:** O(n)
- **Usuwanie:** O(n)

Lista Dwukierunkowa

Opis: Lista, w której każdy węzeł zawiera wskaźnik zarówno na następny, jak i poprzedni węzeł, umożliwiając dwukierunkowe przeglądanie listy.

Implementacja:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None  # Wskaźnik na poprzedni węzeł
        self.next = None  # Wskaźnik na następny węzeł

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
        new_node.prev = last

    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        if self.head:
            self.head.prev = new_node
        self.head = new_node

    def delete(self, node):
        if node.prev:
            node.prev.next = node.next
        else:
            self.head = node.next  # Node jest głową
        if node.next:
            node.next.prev = node.prev

    def display_forward(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        print("Forward:", " <-> ".join(map(str, elements)))

    def display_backward(self):
        elements = []
        current = self.head
        while current and current.next:
            current = current.next
        while current:
            elements.append(current.data)
```

```
            current = current.prev
        print("Backward:", " <-> ".join(map(str, elements)))
```

Przykład Użycia:

```python
dll = DoublyLinkedList()
dll.append(10)
dll.append(20)
dll.prepend(5)
dll.display_forward()   # Wyjście: Forward: 5 <-> 10 <-> 20
dll.display_backward()  # Wyjście: Backward: 20 <-> 10 <-> 5
```

Złożoność Operacji:

- **Dodawanie na końcu lub początku:** O(1)
- **Wyszukiwanie:** O(n)
- **Usuwanie:** O(1) (po znalezieniu węzła)

Lista Tablicowa (ArrayList)

Opis: Lista implementowana jako dynamiczna tablica, która automatycznie zmienia swój rozmiar w razie potrzeby.

Implementacja:

```python
class ArrayList:
    def __init__(self):
        self.array = []

    def append(self, data):
        self.array.append(data)

    def insert(self, index, data):
        self.array.insert(index, data)

    def delete(self, index):
        if index < len(self.array):
            self.array.pop(index)

    def get(self, index):
        if index < len(self.array):
            return self.array[index]
        return None

    def display(self):
        print(self.array)
```

Przykład Użycia:

```
al = ArrayList()
al.append(10)
al.append(20)
al.insert(1, 15)
al.display()  # Wyjście: [10, 15, 20]
al.delete(1)
al.display()  # Wyjście: [10, 20]
```

Złożoność Operacji:

- **Dodawanie na końcu:** O(1) amortyzowane
- **Dodawanie w środku lub początku:** O(n)
- **Wyszukiwanie:** O(1)
- **Usuwanie:** O(n)

Zalety i Wady:

Struktura	Zalety	Wady
Lista Jednokierunkowa	Prosta implementacja, efektywne dodawanie/usuwanie na początku	Trudniejsze do przeglądania wstecz, wyszukiwanie jest O(n)
Lista Dwukierunkowa	Dwukierunkowe przeglądanie, efektywne usuwanie w dowolnym miejscu	Więcej pamięci na wskaźniki, bardziej skomplikowana implementacja
Lista Tablicowa	Szybki dostęp do elementów, prostsza implementacja	Koszty związane z dodawaniem/usuwaniem w środku listy, konieczność dynamicznego zarządzania pamięcią

Stosy

Opis: Stos (ang. Stack) to struktura danych działająca na zasadzie LIFO (Last In, First Out). Oznacza to, że ostatni dodany element jest pierwszym, który zostanie usunięty.

Operacje Podstawowe:

- **Push:** Dodanie elementu na szczyt stosu
- **Pop:** Usunięcie elementu ze szczytu stosu
- **Peek/Top:** Podgląd elementu na szczycie stosu bez usuwania
- **IsEmpty:** Sprawdzenie, czy stos jest pusty

Implementacja:

```
class Stack:
    def __init__(self):
        self.items = []
```

```
    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        raise IndexError("Pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("Peek from empty stack")

    def size(self):
        return len(self.items)

    def display(self):
        print("Stack:", self.items)
```

Przykład Użycia:

```
s = Stack()
s.push(10)
s.push(20)
s.display()  # Wyjście: Stack: [10, 20]
print(s.pop())  # Wyjście: 20
s.display()  # Wyjście: Stack: [10]
```

Złożoność Operacji:

- **Push, Pop, Peek:** O(1)
- **IsEmpty, Size:** O(1)

Zastosowania:

- Implementacja funkcji wywołań rekurencyjnych
- Przeglądanie wyrażeń w notacji odwrotnej polskiej
- Implementacja algorytmów DFS w grafach
- Cofanie operacji w edytorach tekstu

Kolejki

Opis: Kolejka (ang. Queue) to struktura danych działająca na zasadzie FIFO (First In, First Out). Oznacza to, że pierwszy dodany element jest pierwszym, który zostanie usunięty.

Rodzaje Kolejek:

1. **Kolejka Prosta (Simple Queue)**
2. **Kolejka Priorytetowa (Priority Queue)**
3. **Kolejka Dwustronna (Deque)**

Kolejka Prosta

Opis: Najprostsza forma kolejki, gdzie elementy są dodawane na końcu i usuwane z początku.

Implementacja:

```python
from collections import deque

class SimpleQueue:
    def __init__(self):
        self.queue = deque()

    def is_empty(self):
        return len(self.queue) == 0

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()
        raise IndexError("Dequeue from empty queue")

    def front(self):
        if not self.is_empty():
            return self.queue[0]
        raise IndexError("Front from empty queue")

    def size(self):
        return len(self.queue)

    def display(self):
        print("Queue:", list(self.queue))
```

Przykład Użycia:

```python
q = SimpleQueue()
q.enqueue(10)
q.enqueue(20)
q.display()  # Wyjście: Queue: [10, 20]
print(q.dequeue())  # Wyjście: 10
q.display()  # Wyjście: Queue: [20]
```

Złożoność Operacji:

- **Enqueue, Dequeue:** O(1)
- **Front, Size, IsEmpty:** O(1)

Kolejka Priorytetowa

Opis: Kolejka, w której każdy element ma przypisany priorytet. Element z najwyższym priorytetem jest usuwany jako pierwszy.

Implementacja:

```python
import heapq

class PriorityQueue:
    def __init__(self):
        self.heap = []

    def is_empty(self):
        return len(self.heap) == 0

    def enqueue(self, item, priority):
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        if not self.is_empty():
            return heapq.heappop(self.heap)[1]
        raise IndexError("Dequeue from empty priority queue")

    def peek(self):
        if not self.is_empty():
            return self.heap[0][1]
        raise IndexError("Peek from empty priority queue")

    def size(self):
        return len(self.heap)

    def display(self):
        print("Priority Queue:", self.heap)
```

Przykład Użycia:

```python
pq = PriorityQueue()
pq.enqueue("task1", 2)
pq.enqueue("task2", 1)
pq.enqueue("task3", 3)
pq.display()  # Wyjście: Priority Queue: [(1, 'task2'), (2, 'task1'), (3,
'task3')]
print(pq.dequeue())  # Wyjście: task2
pq.display()  # Wyjście: Priority Queue: [(2, 'task1'), (3, 'task3')]
```

Złożoność Operacji:

- **Enqueue, Dequeue:** O(log n)
- **Peek, Size, IsEmpty:** O(1)

Zastosowania:

- Harmonogramowanie zadań w systemach operacyjnych
- Algorytm Dijkstry do znajdowania najkrótszej ścieżki
- Implementacja algorytmów Huffmana

Kolejka Dwustronna (Deque)

Opis: Kolejka dwustronna (Deque) umożliwia dodawanie i usuwanie elementów zarówno z początku, jak i z końca kolejki.

Implementacja:

```python
from collections import deque

class Deque:
    def __init__(self):
        self.deque = deque()

    def is_empty(self):
        return len(self.deque) == 0

    def add_front(self, item):
        self.deque.appendleft(item)

    def add_rear(self, item):
        self.deque.append(item)

    def remove_front(self):
        if not self.is_empty():
            return self.deque.popleft()
        raise IndexError("Remove from empty deque")

    def remove_rear(self):
        if not self.is_empty():
            return self.deque.pop()
        raise IndexError("Remove from empty deque")

    def front(self):
        if not self.is_empty():
            return self.deque[0]
        raise IndexError("Front from empty deque")

    def rear(self):
        if not self.is_empty():
            return self.deque[-1]
        raise IndexError("Rear from empty deque")
```

```python
    def size(self):
        return len(self.deque)

    def display(self):
        print("Deque:", list(self.deque))
```

Przykład Użycia:

```python
dq = Deque()
dq.add_rear(10)
dq.add_front(20)
dq.display()  # Wyjście: Deque: [20, 10]
print(dq.remove_rear())  # Wyjście: 10
dq.display()  # Wyjście: Deque: [20]
```

Złożoność Operacji:

- **Dodawanie/Usuwanie z przodu i tyłu:** O(1)
- **Dostęp do przednich i tylnych elementów:** O(1)
- **Rozmiar, IsEmpty:** O(1)

Zastosowania:

- Implementacja algorytmów BFS i DFS
- Przechowywanie historii odwiedzin (np. w przeglądarkach internetowych)
- Realizacja struktur danych takich jak stos i kolejka w jednym

Grafy

Opis: Graf to struktura danych składająca się z wierzchołków (nóg) oraz krawędzi łączących te wierzchołki. Grafy mogą być skierowane lub nieskierowane, ważone lub nieważone.

Rodzaje Grafów:

- **Graf Nieskierowany (Undirected Graph)**
- **Graf Skierowany (Directed Graph)**
- **Graf Ważony (Weighted Graph)**
- **Graf Niespójny (Disconnected Graph)**
- **Graf Spójny (Connected Graph)**
- **Graf Cycliczny (Cyclic Graph)**
- **Graf Acycliczny (Acyclic Graph)**

Reprezentacja Grafów

Grafy można reprezentować na kilka sposobów, z których najpopularniejsze to:

1. **Lista Sąsiedztwa (Adjacency List)**
2. **Macierz Sąsiedztwa (Adjacency Matrix)**

3. **Lista Krawędzi (Edge List)**

Lista Sąsiedztwa:

- Przechowuje dla każdego wierzchołka listę sąsiadów.
- Oszczędna w przypadku grafów rzadkich.

Macierz Sąsiedztwa:

- Dwuwymiarowa tablica, gdzie komórka (i, j) oznacza obecność (lub wagę) krawędzi między wierzchołkami i i j.
- Wymaga $O(V^2)$ pamięci, gdzie V to liczba wierzchołków.

Lista Krawędzi:

- Lista par (lub trójek dla grafów ważonych) reprezentujących krawędzie.
- Przydatna do algorytmów przeglądających wszystkie krawędzie.

Implementacje Grafów

Lista Sąsiedztwa

Implementacja:

```
class GraphAdjacencyList:
    def __init__(self, directed=False):
        self.adj_list = {}
        self.directed = directed

    def add_vertex(self, vertex):
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, src, dest, weight=1):
        self.add_vertex(src)
        self.add_vertex(dest)
        self.adj_list[src].append((dest, weight))
        if not self.directed:
            self.adj_list[dest].append((src, weight))

    def remove_edge(self, src, dest):
        if src in self.adj_list:
            self.adj_list[src] = [ (d, w) for (d, w) in self.adj_list[src] if d != dest ]
        if not self.directed and dest in self.adj_list:
            self.adj_list[dest] = [ (s, w) for (s, w) in self.adj_list[dest] if s != src ]

    def display(self):
        for vertex in self.adj_list:
            print(f"{vertex}: {self.adj_list[vertex]}")
```

Przykład Użycia:

```
g = GraphAdjacencyList()
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'C', 2)
g.add_edge('C', 'D')
g.display()
# Wyjście:
# A: [('B', 1), ('C', 1)]
# B: [('A', 1), ('C', 2)]
# C: [('A', 1), ('B', 2), ('D', 1)]
# D: [('C', 1)]
```

Zalety:

- Efektywna pamięciowo dla grafów rzadkich.
- Szybki dostęp do listy sąsiadów.

Wady:

- Sprawdzanie istnienia konkretnej krawędzi może wymagać przeszukania listy.

Macierz Sąsiedztwa

Implementacja:

```
class GraphAdjacencyMatrix:
    def __init__(self, vertices, directed=False):
        self.V = vertices
        self.directed = directed
        self.matrix = [ [0]*vertices for _ in range(vertices) ]
        self.vertex_map = {i: chr(65+i) for i in range(vertices)}  # Przykładowa mapowanie 0->'A', 1->'B', etc.
        self.reverse_map = {v: k for k, v in self.vertex_map.items()}

    def add_edge(self, src, dest, weight=1):
        src_idx = self.reverse_map[src]
        dest_idx = self.reverse_map[dest]
        self.matrix[src_idx][dest_idx] = weight
        if not self.directed:
            self.matrix[dest_idx][src_idx] = weight

    def remove_edge(self, src, dest):
        src_idx = self.reverse_map[src]
        dest_idx = self.reverse_map[dest]
        self.matrix[src_idx][dest_idx] = 0
        if not self.directed:
            self.matrix[dest_idx][src_idx] = 0
```

```python
    def display(self):
        print(" ", end=" ")
        for v in self.vertex_map.values():
            print(v, end=" ")
        print()
        for i in range(self.V):
            print(self.vertex_map[i], end=" ")
            for j in range(self.V):
                print(self.matrix[i][j], end=" ")
            print()
```

Przykład Użycia:

```python
g = GraphAdjacencyMatrix(4)
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'C')
g.add_edge('C', 'D')
g.display()
# Wyjście:
#   A B C D
# A 0 1 1 0
# B 1 0 1 0
# C 1 1 0 1
# D 0 0 1 0
```

Zalety:

- Szybkie sprawdzanie istnienia krawędzi (O(1)).
- Prosta implementacja.

Wady:

- Zużywa dużo pamięci dla dużych, rzadkich grafów.
- Słaba wydajność dla operacji iteracji nad sąsiadami.

Lista Krawędzi

Opis: Lista przechowuje wszystkie krawędzie grafu jako pary (lub trójki dla grafów ważonych).

Implementacja:

```python
class GraphEdgeList:
    def __init__(self, directed=False):
        self.edges = []
        self.directed = directed

    def add_edge(self, src, dest, weight=1):
        self.edges.append((src, dest, weight))
```

```
        if not self.directed:
            self.edges.append((dest, src, weight))

    def remove_edge(self, src, dest):
        self.edges = [ edge for edge in self.edges if not (edge[0] == src and
edge[1] == dest) ]
        if not self.directed:
            self.edges = [ edge for edge in self.edges if not (edge[0] == dest and
edge[1] == src) ]

    def display(self):
        for edge in self.edges:
            print(edge)
```

Przykład Użycia:

```
g = GraphEdgeList()
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'C')
g.add_edge('C', 'D')
g.display()
# Wyjście:
# ('A', 'B', 1)
# ('A', 'C', 1)
# ('B', 'C', 1)
# ('C', 'A', 1)
# ('C', 'B', 1)
# ('C', 'D', 1)
# ('D', 'C', 1)
```

Zalety:

- Prosta reprezentacja, szczególnie dla grafów rzadkich.
- Łatwość iteracji nad wszystkimi krawędziami.

Wady:

- Sprawdzanie istnienia konkretnej krawędzi jest kosztowne (O(n)).
- Nieefektywne dla grafów gęstych.

Drzewa

Opis: Drzewo to struktura danych, która reprezentuje hierarchię. Składa się z węzłów połączonych krawędziami, gdzie każdy węzeł ma jednego rodzica (z wyjątkiem korzenia) i zero lub więcej dzieci.

Rodzaje Drzew:

1. **Drzewa Binarne (Binary Trees)**
2. **Drzewa AVL (AVL Trees)**

3. **Drzewa Czerwono-Czarne (Red-Black Trees)**
4. **Drzewa B-drzewa (B-Trees)**
5. **Drzewa Trójkowe (Tries)**
6. **Drzewa Segmentowe (Segment Trees)**

Drzewa Binarne

Opis: Drzewo, w którym każdy węzeł ma co najwyżej dwóch potomków, zazwyczaj określanych jako lewy i prawy.

Podtypy:

- **Drzewo Binarne Poszukiwań (Binary Search Tree - BST)**
- **Drzewo Binarne Równoważone (Balanced Binary Tree)**
- **Pełne Drzewo Binarne (Full Binary Tree)**
- **Pełne Drzewo Binarne (Complete Binary Tree)**

Implementacja Drzewa Binarnego:

```python
class TreeNode:
    def __init__(self, key):
        self.left = None  # Lewy potomek
        self.right = None  # Prawy potomek
        self.val = key  # Wartość węzła

def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.val, end=' ')
        inorder_traversal(root.right)

def preorder_traversal(root):
    if root:
        print(root.val, end=' ')
        preorder_traversal(root.left)
        preorder_traversal(root.right)

def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.val, end=' ')
```

Przykład Użycia:

```python
root = TreeNode(10)
root.left = TreeNode(5)
root.right = TreeNode(15)
root.left.left = TreeNode(2)
```

```
root.left.right = TreeNode(7)
root.right.left = TreeNode(12)
root.right.right = TreeNode(20)

print("Inorder Traversal:")
inorder_traversal(root)  # Wyjście: 2 5 7 10 12 15 20

print("\nPreorder Traversal:")
preorder_traversal(root)  # Wyjście: 10 5 2 7 15 12 20

print("\nPostorder Traversal:")
postorder_traversal(root)  # Wyjście: 2 7 5 12 20 15 10
```

Złożoność Operacji:

- **Wstawianie, Wyszukiwanie, Usuwanie:** O(h), gdzie h to wysokość drzewa (najgorszy przypadek O(n))
- **Traversale (Przejścia):** O(n)

Drzewa AVL

Opis: Drzewa AVL to samobalansujące się drzewa binarne poszukiwań, w których dla każdego węzła różnica wysokości lewego i prawego poddrzewa nie przekracza 1. Zapewnia to złożoność operacji O(log n).

Implementacja:

```python
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def insert(self, root, key):
        # Krok 1: Normalne wstawianie
        if not root:
            return AVLNode(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # Krok 2: Aktualizacja wysokości
        root.height = 1 + max(self.get_height(root.left),
                              self.get_height(root.right))

        # Krok 3: Sprawdzenie równowagi
        balance = self.get_balance(root)

        # Krok 4: Rotacje w razie potrzeby
        # Lewo Lewo
```

```
        if balance > 1 and key < root.left.key:
            return self.right_rotate(root)

        # Prawo Prawo
        if balance < -1 and key > root.right.key:
            return self.left_rotate(root)

        # Lewo Prawo
        if balance > 1 and key > root.left.key:
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)

        # Prawo Lewo
        if balance < -1 and key < root.right.key:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)

        return root

    def left_rotate(self, z):
        y = z.right
        T2 = y.left

        # Rotacja
        y.left = z
        z.right = T2

        # Aktualizacja wysokości
        z.height = 1 + max(self.get_height(z.left),
                           self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left),
                           self.get_height(y.right))

        return y

    def right_rotate(self, z):
        y = z.left
        T3 = y.right

        # Rotacja
        y.right = z
        z.left = T3

        # Aktualizacja wysokości
        z.height = 1 + max(self.get_height(z.left),
                           self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left),
                           self.get_height(y.right))

        return y

    def get_height(self, node):
        if not node:
            return 0
```

```
        return node.height

    def get_balance(self, node):
        if not node:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    def inorder_traversal(self, root):
        if root:
            self.inorder_traversal(root.left)
            print(root.key, end=' ')
            self.inorder_traversal(root.right)
```

Przykład Użycia:

```
avl = AVLTree()
root = None
keys = [10, 20, 30, 40, 50, 25]

for key in keys:
    root = avl.insert(root, key)

avl.inorder_traversal(root)  # Wyjście: 10 20 25 30 40 50
```

Złożoność Operacji:

- **Wstawianie, Usuwanie, Wyszukiwanie:** O(log n)

Zastosowania:

- Bazy danych
- Systemy plików
- Systemy operacyjne

Drzewa Czerwono-Czarne

Opis: Drzewa czerwono-czarne to samobalansujące się drzewa binarne poszukiwań, które zapewniają złożoność operacji O(log n) poprzez stosowanie reguł kolorowania węzłów (czerwony lub czarny).

Reguły Drzew Czerwono-Czarnych:

1. Każdy węzeł jest czerwony lub czarny.
2. Korzeń jest czarny.
3. Wszystkie liście (NIL) są czarne.
4. Czerwony węzeł nie może mieć czerwonych dzieci.
5. Każda ścieżka od węzła do liścia zawiera tę samą liczbę czarnych węzłów.

Implementacja: Implementacja drzewa czerwono-czarnego jest bardziej skomplikowana niż AVL i wymaga obsługi dodatkowych przypadków rotacji oraz kolorowania.

Przykład Implementacji: Ze względu na złożoność implementacji, poniżej znajduje się uproszczony pseudokod.

```python
class RBNode:
    def __init__(self, key, color='red'):
        self.key = key
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    def __init__(self):
        self.NIL = RBNode(key=None, color='black')  # Liść
        self.root = self.NIL

    def insert(self, key):
        # Implementacja wstawiania z korekcją kolorów i rotacjami
        pass  # Ze względu na złożoność pomijamy szczegóły implementacji

    def inorder_traversal(self, node):
        if node != self.NIL:
            self.inorder_traversal(node.left)
            print(node.key, end=' ')
            self.inorder_traversal(node.right)
```

Złożoność Operacji:

- **Wstawianie, Usuwanie, Wyszukiwanie:** O(log n)

Zastosowania:

- Bazy danych
- Implementacje słowników i zbiorów
- Systemy plików

Drzewa B-drzewa

Opis: Drzewa B-drzewa to samobalansujące się, wielokierunkowe drzewa poszukiwań binarnych, które są szeroko stosowane w systemach baz danych i systemach plików ze względu na efektywne zarządzanie dużymi zbiorami danych.

Cechy Drzew B:

- Każdy węzeł może mieć więcej niż dwóch potomków.
- Wszystkie liście znajdują się na tym samym poziomie.
- Liczba kluczy w każdym węźle jest ograniczona przedziałem.

Implementacja: Implementacja drzewa B jest złożona i zwykle wymaga szczegółowej obsługi różnych przypadków rotacji i podziałów węzłów.

Przykład Implementacji: Ze względu na złożoność implementacji, poniżej znajduje się uproszczony pseudokod.

```
class BTreeNode:
    def __init__(self, t, leaf=False):
        self.t = t  # Minimalny stopień
        self.leaf = leaf
        self.keys = []
        self.children = []

class BTree:
    def __init__(self, t):
        self.root = BTreeNode(t, leaf=True)
        self.t = t

    def insert(self, key):
        # Implementacja wstawiania z podziałem węzłów
        pass  # Ze względu na złożoność pomijamy szczegóły implementacji

    def traverse(self, node=None):
        if node is None:
            node = self.root
        for i in range(len(node.keys)):
            if not node.leaf:
                self.traverse(node.children[i])
            print(node.keys[i], end=' ')
        if not node.leaf:
            self.traverse(node.children[len(node.keys)])
```

Złożoność Operacji:

- **Wstawianie, Usuwanie, Wyszukiwanie:** O(log n)

Zastosowania:

- Systemy zarządzania bazami danych (DBMS)
- Systemy plików (np. NTFS, HFS+)

Słowniki (Mapy)

Opis: Słownik (ang. Dictionary lub Map) to struktura danych przechowująca pary klucz-wartość, umożliwiająca szybki dostęp do wartości na podstawie unikalnego klucza.

Operacje Podstawowe:

- **Insert/Add:** Dodanie pary klucz-wartość
- **Delete/Remove:** Usunięcie pary na podstawie klucza
- **Search/Get:** Pobranie wartości na podstawie klucza
- **Update:** Aktualizacja wartości dla danego klucza

Implementacje Słowników:

1. **Tablica Hashująca (Hash Table)**
2. **Drzewo Binarne Poszukiwań (BST)**
3. **Drzewo Czerwono-Czarne**
4. **Drzewo Trie**

Tablica Hashująca

Opis: Tablica hashująca przechowuje dane w oparciu o funkcję hashującą, która mapuje klucze na indeksy tablicy.

Funkcje Hashujące:

- **Funkcje deterministyczne:** Te same klucze zawsze dają ten sam hash.
- **Minimalizacja kolizji:** Rozkład wartości hash powinien być równomierny.

Implementacja:

```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        hash_key = self.hash_function(key)
        for idx, (k, v) in enumerate(self.table[hash_key]):
            if k == key:
                self.table[hash_key][idx] = (key, value)  # Aktualizacja wartości
                return
        self.table[hash_key].append((key, value))

    def get(self, key):
        hash_key = self.hash_function(key)
        for (k, v) in self.table[hash_key]:
            if k == key:
                return v
        raise KeyError(f"{key} not found")

    def delete(self, key):
        hash_key = self.hash_function(key)
        for idx, (k, v) in enumerate(self.table[hash_key]):
            if k == key:
                del self.table[hash_key][idx]
                return
        raise KeyError(f"{key} not found")

    def display(self):
        for idx, bucket in enumerate(self.table):
            print(f"Bucket {idx}: {bucket}")
```

Przykład Użycia:

```
ht = HashTable()
ht.insert("apple", 1)
ht.insert("banana", 2)
ht.insert("orange", 3)
ht.display()
# Wyjście:
# Bucket 0: []
# Bucket 1: []
# ...
# Bucket 2: [('banana', 2)]
# Bucket 5: [('apple', 1)]
# Bucket 7: [('orange', 3)]
# ...
print(ht.get("banana"))  # Wyjście: 2
ht.delete("banana")
ht.display()
# Wyjście:
# Bucket 2: []
```

Złożoność Operacji:

- **Insert, Delete, Get:** O(1) średnio, O(n) w najgorszym przypadku (kolizje)

Zalety:

- Szybki dostęp do danych
- Efektywna pamięciowo dla dużych zbiorów danych

Wady:

- Kolizje mogą wpływać na wydajność
- Wymaga dobrze dobranej funkcji hashującej

Drzewo Trie

Opis: Trie (lub drzewo prefiksowe) to drzewo specjalnego typu używane głównie do przechowywania zbiorów słów, umożliwiające efektywne wyszukiwanie i operacje na prefiksach.

Implementacja:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
```

```python
        self.root = TrieNode()

    def insert(self, word):
        current = self.root
        for char in word:
            if char not in current.children:
                current.children[char] = TrieNode()
            current = current.children[char]
        current.is_end_of_word = True

    def search(self, word):
        current = self.root
        for char in word:
            if char not in current.children:
                return False
            current = current.children[char]
        return current.is_end_of_word

    def starts_with(self, prefix):
        current = self.root
        for char in prefix:
            if char not in current.children:
                return False
            current = current.children[char]
        return True

    def display(self, node=None, word=''):
        if node is None:
            node = self.root
        if node.is_end_of_word:
            print(word)
        for char, child in node.children.items():
            self.display(child, word + char)
```

Przykład Użycia:

```python
trie = Trie()
words = ["apple", "app", "apricot", "banana"]
for word in words:
    trie.insert(word)

print(trie.search("app"))       # Wyjście: True
print(trie.search("appl"))      # Wyjście: False
print(trie.starts_with("apr"))  # Wyjście: True

print("All words in trie:")
trie.display()
# Wyjście:
# app
# apple
# apricot
# banana
```

```
```

Złożoność Operacji:

- **Insert, Search, StartsWith:** O(m), gdzie m to długość słowa

Zastosowania:

- Autouzupełnianie
- Słowniki online
- Kompresja danych

Drzewa Poszukiwań Binarnych (BST)

Opis: Drzewo poszukiwań binarnych (BST) to drzewo binarne, w którym dla każdego węzła wszystkie klucze w lewym poddrzewie są mniejsze, a w prawym większe od klucza w węźle.

Implementacja BST:

```python
class BSTNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = BSTNode(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = BSTNode(key)
            else:
                self._insert_recursive(node.left, key)
        else:
            if node.right is None:
                node.right = BSTNode(key)
            else:
                self._insert_recursive(node.right, key)

    def search(self, key):
        return self._search_recursive(self.root, key)

    def _search_recursive(self, node, key):
        if node is None or node.val == key:
```

```
            return node
        if key < node.val:
            return self._search_recursive(node.left, key)
        return self._search_recursive(node.right, key)

    def inorder_traversal(self, node=None):
        if node is None:
            node = self.root
        if node:
            self.inorder_traversal(node.left)
            print(node.val, end=' ')
            self.inorder_traversal(node.right)

    def delete(self, key):
        self.root = self._delete_recursive(self.root, key)

    def _delete_recursive(self, node, key):
        if node is None:
            return node
        if key < node.val:
            node.left = self._delete_recursive(node.left, key)
        elif key > node.val:
            node.right = self._delete_recursive(node.right, key)
        else:
            # Węzeł z jednym lub bez potomków
            if node.left is None:
                temp = node.right
                node = None
                return temp
            elif node.right is None:
                temp = node.left
                node = None
                return temp
            # Węzeł z dwoma potomkami: Znajdź inorder successor
            temp = self._min_value_node(node.right)
            node.val = temp.val
            node.right = self._delete_recursive(node.right, temp.val)
        return node

    def _min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current
```

Przykład Użycia:

```
bst = BinarySearchTree()
keys = [50, 30, 70, 20, 40, 60, 80]
for key in keys:
    bst.insert(key)
```

```
print("Inorder Traversal:")
bst.inorder_traversal()  # Wyjście: 20 30 40 50 60 70 80

print("\nSearch for 40:", bst.search(40) is not None)  # Wyjście: True
print("Search for 25:", bst.search(25) is not None)    # Wyjście: False

bst.delete(20)
bst.delete(30)
bst.delete(50)
print("Inorder Traversal after deletions:")
bst.inorder_traversal()  # Wyjście: 40 60 70 80
```

Złożoność Operacji:

- **Wstawianie, Wyszukiwanie, Usuwanie:** O(h), gdzie h to wysokość drzewa (najgorszy przypadek O(n))

Zastosowania:

- Implementacja słowników i zbiorów
- Przechowywanie danych w sposób uporządkowany
- Wykorzystywane w algorytmach grafowych

Algorytmy Sortowania

Opis: Sortowanie to proces uporządkowywania elementów w określonym porządku (rosnącym lub malejącym). Wybór odpowiedniego algorytmu sortowania zależy od rozmiaru danych, stabilności sortowania, złożoności czasowej i pamięciowej oraz innych czynników.

Kluczowe Wymagania:

- **Stabilność:** Czy elementy o równych kluczach zachowują swoją kolejność.
- **Złożoność czasowa:** Średnia, najlepsza i najgorsza złożoność.
- **Złożoność pamięciowa:** Ilość dodatkowej pamięci potrzebnej do sortowania.
- **Przystosowanie do różnych typów danych:** Czy algorytm działa na różnych strukturach danych.

Poniżej opisujemy różne algorytmy sortowania, ich rodzaje, przykłady, złożoność czasową oraz schematy blokowe.

Sortowanie Bąbelkowe (Bubble Sort)

Opis: Sortowanie bąbelkowe to prosty algorytm sortowania, który wielokrotnie przechodzi przez listę, porównując sąsiednie elementy i zamieniając je, jeśli są w niewłaściwej kolejności. Proces ten powtarza się do momentu, gdy lista jest w pełni posortowana.

Złożoność:

- **Średnia:** O(n^2)
- **Najlepsza:** O(n) (jeśli lista jest już posortowana)
- **Najgorsza:** O(n^2)

Implementacja:

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False  # Flaga do optymalizacji
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]  # Zamiana miejscami
                swapped = True
        if not swapped:
            break  # Jeśli nie było zamian, lista jest posortowana
```

Przykład Użycia:

```python
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array is:", arr)  # Wyjście: Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

Schemat Blokowy:

```
[Start]
   ↓
[For i from 0 to n-1]
   ↓
[Set swapped = False]
   ↓
[For j from 0 to n-i-2]
   ↓
[If arr[j] > arr[j+1]]
   ↓
[Swap arr[j] and arr[j+1]]
   ↓
[Set swapped = True]
   ↓
[End Inner Loop]
   ↓
[If swapped == False]
   ↓
[Break]
   ↓
[End Outer Loop]
   ↓
[Stop]
```

Zalety:

- Prosta implementacja
- Stabilny (zachowuje kolejność równych elementów)

Wady:

- Niska wydajność dla dużych zbiorów danych
- Nieefektywne porównania i zamiany

Zastosowania:

- Edukacyjne, do nauki podstaw sortowania
- Małe, niemal posortowane listy

Sortowanie Przez Wstawianie (Insertion Sort)

Opis: Sortowanie przez wstawianie buduje posortowaną listę jeden element na raz, wstawiając każdy element w odpowiednie miejsce w już posortowanej części listy.

Złożoność:

- **Średnia:** $O(n^2)$
- **Najlepsza:** $O(n)$ (jeśli lista jest już posortowana)
- **Najgorsza:** $O(n^2)$

Implementacja:

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]  # Element do wstawienia
        j = i - 1
        # Przesuwanie elementów większych od key w prawo
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key  # Wstawienie key w odpowiednie miejsce
```

Przykład Użycia:

```python
arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)  # Wyjście: Sorted array is: [5, 6, 11, 12, 13]
```

Schemat Blokowy:

```
[Start]
   ↓
[For i from 1 to n-1]
   ↓
```

```
[Set key = arr[i]]
   ↓
[j = i - 1]
   ↓
[While j >=0 and key < arr[j]]
   ↓
[Move arr[j] to arr[j+1]]
   ↓
[j = j -1]
   ↓
[Set arr[j+1] = key]
   ↓
[End While]
   ↓
[End For]
   ↓
[Stop]
```

Zalety:

- Stabilny
- Działa dobrze dla małych lub prawie posortowanych list
- Łatwy do implementacji

Wady:

- Niska wydajność dla dużych list
- Nieefektywne w przypadku dużych, losowych zbiorów danych

Zastosowania:

- Sortowanie małych tablic
- Jako część bardziej złożonych algorytmów (np. sortowanie hybrydowe)

Sortowanie Przez Wybor (Selection Sort)

Opis: Sortowanie przez wybór działa poprzez iteracyjne znajdowanie najmniejszego (lub największego) elementu z nieposortowanej części listy i zamianę go z pierwszym elementem tej części.

Złożoność:

- **Średnia:** $O(n^2)$
- **Najlepsza:** $O(n^2)$
- **Najgorsza:** $O(n^2)$

Implementacja:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i  # Indeks najmniejszego elementu
```

```
        for j in range(i+1, n):
            if arr[min_idx] > arr[j]:
                min_idx = j  # Aktualizacja indeksu najmniejszego
        arr[i], arr[min_idx] = arr[min_idx], arr[i]  # Zamiana miejscami
```

Przykład Użycia:

```
arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Sorted array is:", arr)  # Wyjście: Sorted array is: [11, 12, 22, 25, 64]
```

Schemat Blokowy:

```
[Start]
   ↓
[For i from 0 to n-1]
   ↓
[Set min_idx = i]
   ↓
[For j from i+1 to n-1]
       ↓
    [If arr[j] < arr[min_idx]]
       ↓
    [Set min_idx = j]
       ↓
[End Inner Loop]
   ↓
[Swap arr[i] and arr[min_idx]]
   ↓
[End Outer Loop]
   ↓
[Stop]
```

Zalety:

- Prosta implementacja
- Stała złożoność czasowa niezależnie od danych

Wady:

- Nie jest stabilny
- Niska wydajność dla dużych list

Zastosowania:

- Małe, statyczne listy
- Edukacyjne, do nauki podstaw sortowania

Sortowanie Szybkie (Quick Sort)

Opis: Sortowanie szybkie to algorytm sortowania typu "dziel i zwyciężaj". Wybiera pivot, dzieli listę na elementy mniejsze i większe od pivota, a następnie rekurencyjnie sortuje podlisty.

Złożoność:

- **Średnia:** O(n log n)
- **Najlepsza:** O(n log n)
- **Najgorsza:** $O(n^2)$ (gdy pivot jest zawsze największy lub najmniejszy)

Implementacja:

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]  # Wybór pivota (środkowy element)
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quick_sort(left) + middle + quick_sort(right)
```

Przykład Użycia:

```python
arr = [10, 7, 8, 9, 1, 5]
sorted_arr = quick_sort(arr)
print("Sorted array is:", sorted_arr)  # Wyjście: Sorted array is: [1, 5, 7, 8, 9, 10]
```

Schemat Blokowy:

```
[Start]
   ↓
[If len(arr) <= 1]
   ↓
[Return arr]
   ↓
[Choose pivot]
   ↓
[Partition arr into left, middle, right]
   ↓
[Recurse on left and right]
   ↓
[Combine results]
   ↓
[Stop]
```

Zalety:

- Średnio bardzo szybki
- Niska złożoność pamięciowa

Wady:

- Niewydajny w najgorszym przypadku
- Niezależnie od implementacji, nie jest stabilny

Zastosowania:

- Ogólne sortowanie danych
- Szybkie sortowanie dużych zestawów danych w pamięci

Sortowanie Scalanie (Merge Sort)

Opis: Sortowanie scalanie to algorytm sortowania typu "dziel i zwyciężaj". Dzieli listę na pół, sortuje każdą połowę rekurencyjnie, a następnie scala posortowane listy.

Złożoność:

- **Średnia:** O(n log n)
- **Najlepsza:** O(n log n)
- **Najgorsza:** O(n log n)

Implementacja:

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2  # Znalezienie środka listy
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)  # Sortowanie lewej połowy
        merge_sort(R)  # Sortowanie prawej połowy

        i = j = k = 0

        # Scalanie posortowanych list
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Kopiowanie pozostałych elementów L
        while i < len(L):
            arr[k] = L[i]
```

```
            i += 1
            k += 1

        # Kopiowanie pozostałych elementów R
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Przykład Użycia:

```
arr = [12, 11, 13, 5, 6, 7]
merge_sort(arr)
print("Sorted array is:", arr)  # Wyjście: Sorted array is: [5, 6, 7, 11, 12, 13]
```

Schemat Blokowy:

```
[Start]
   ↓
[If len(arr) > 1]
   ↓
[Divide arr into L and R]
   ↓
[Recursively sort L and R]
   ↓
[Initialize i, j, k = 0]
   ↓
[While i < len(L) and j < len(R)]
       ↓
    [Compare L[i] and R[j]]
       ↓
    [Move smaller to arr[k]]
       ↓
    [Increment i/j and k]
   ↓
[While i < len(L)]
   ↓
[Move remaining L[i] to arr[k]]
   ↓
[While j < len(R)]
   ↓
[Move remaining R[j] to arr[k]]
   ↓
[Stop]
```

Zalety:

- Stabilny

- Gwarantowana złożoność czasowa O(n log n)
- Efektywny dla dużych danych

Wady:

- Wymaga dodatkowej pamięci O(n)
- Mniej efektywny dla małych list

Zastosowania:

- Sortowanie danych w systemach zewnętrznych (np. na dyskach)
- Algorytmy równoległe i wielowątkowe

Sortowanie Kopcowe (Heap Sort)

Opis: Sortowanie kopcowe wykorzystuje strukturę danych zwaną kopcem (heap). Najpierw buduje kopiec maksymalny z listy, a następnie iteracyjnie wyjmuje największy element (korzeń kopca) i umieszcza go na końcu listy.

Złożoność:

- **Średnia:** O(n log n)
- **Najlepsza:** O(n log n)
- **Najgorsza:** O(n log n)

Implementacja:

```
def heapify(arr, n, i):
    largest = i  # Inicjalizacja największego jako korzeń
    l = 2 * i + 1  # Lewy potomek
    r = 2 * i + 2  # Prawy potomek

    # Jeśli lewy potomek jest większy od korzenia
    if l < n and arr[l] > arr[largest]:
        largest = l

    # Jeśli prawy potomek jest większy od obecnego największego
    if r < n and arr[r] > arr[largest]:
        largest = r

    # Jeśli największy nie jest korzeniem
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Zamiana
        heapify(arr, n, largest)  # Rekurencyjne heapify

def heap_sort(arr):
    n = len(arr)

    # Budowanie kopca (rearrange array)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
```

```
    # Wyjmowanie elementów z kopca
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Zamiana
        heapify(arr, i, 0)  # Heapify zmniejszonego kopca
```

Przykład Użycia:

```
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print("Sorted array is:", arr)  # Wyjście: Sorted array is: [5, 6, 7, 11, 12, 13]
```

Schemat Blokowy:

```
[Start]
   ↓
[Build max heap]
   ↓
[For i from n-1 downto 1]
   ↓
[Swap arr[0] with arr[i]]
   ↓
[Heapify reduced heap]
   ↓
[End For]
   ↓
[Stop]
```

Zalety:

- Stała złożoność czasowa O(n log n)
- Niska złożoność pamięciowa (sortowanie w miejscu)

Wady:

- Nie jest stabilny
- Nie jest optymalny dla danych niemal posortowanych

Zastosowania:

- Implementacje sortowania w systemach operacyjnych
- Algorytmy grafowe (np. algorytm Dijkstry)

Sortowanie Radix (Radix Sort)

Opis: Sortowanie Radix to algorytm sortowania nieporównawczego, który sortuje liczby (lub inne dane) poprzez sortowanie ich cyfr od najmniej znaczącej do najbardziej znaczącej. Używa się go zazwyczaj w połączeniu z sortowaniem kubełkowym (Counting Sort).

Złożoność:

- **Średnia:** O(nk), gdzie k to liczba cyfr
- **Najlepsza:** O(nk)
- **Najgorsza:** O(nk)

Implementacja:

```
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n  # Wynikowy posortowany array
    count = [0] * 10  # Licznik dla każdej cyfry 0-9

    # Liczenie wystąpień cyfr
    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    # Zmiana count[i] tak, aby count[i] zawierało pozycję tej cyfry w output
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Budowanie output
    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    # Kopiowanie wyników do arr
    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    # Znalezienie maksymalnej liczby, aby określić liczbę cyfr
    max1 = max(arr)
    exp = 1  # Początkowo sortujemy po najmniej znaczącej cyfrze
    while max1 // exp > 0:
        counting_sort(arr, exp)
        exp *= 10
```

Przykład Użycia:

```
arr = [170, 45, 75, 90, 802, 24, 2, 66]
radix_sort(arr)
print("Sorted array is:", arr)  # Wyjście: Sorted array is: [2, 24, 45, 66, 75, 90, 170, 802]
```

Schemat Blokowy:

```
[Start]
   ↓
[Find the maximum number to know number of digits]
   ↓
[Initialize exp = 1]
   ↓
[While max1 // exp > 0]
       ↓
    [Perform counting sort based on digit at exp]
       ↓
    [Multiply exp by 10]
   ↓
[End While]
   ↓
[Stop]
```

Zalety:

- Bardzo szybki dla liczb o ograniczonej liczbie cyfr
- Stabilny
- Nie wymaga dodatkowej pamięci dla większości implementacji

Wady:

- Ograniczony do danych, które mogą być rozłożone na cyfry (liczby, tekst)
- Złożoność zależy od liczby cyfr

Zastosowania:

- Sortowanie liczb całkowitych
- Przetwarzanie tekstu (np. sortowanie słów w alfabetycznym porządku)

Sortowanie Kubełkowe (Bucket Sort)

Opis: Sortowanie kubełkowe to algorytm sortowania, który dzieli dane na kilka kubełków (bucketów), sortuje każdy kubełek osobno (najczęściej używając innego algorytmu sortowania) i następnie scala posortowane kubełki.

Złożoność:

- **Średnia:** O(n + k), gdzie k to liczba kubełków
- **Najlepsza:** O(n + k)
- **Najgorsza:** O(n^2) (gdy wszystkie elementy trafiają do jednego kubełka)

Implementacja:

```python
def bucket_sort(arr):
    if len(arr) == 0:
```

```
        return arr

    # Znalezienie minimalnej i maksymalnej wartości
    min_val = min(arr)
    max_val = max(arr)

    # Inicjalizacja kubełków
    bucket_count = 10
    buckets = [[] for _ in range(bucket_count)]

    # Dystrybucja elementów do kubełków
    for num in arr:
        index = int(bucket_count * (num - min_val) / (max_val - min_val + 1))
        buckets[index].append(num)

    # Sortowanie każdego kubełka i scalanie wyników
    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(sorted(bucket))  # Można użyć innego algorytmu
sortowania
    return sorted_arr
```

Przykład Użycia:

```
arr = [0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434]
sorted_arr = bucket_sort(arr)
print("Sorted array is:", sorted_arr)
# Wyjście: Sorted array is: [0.1234, 0.3434, 0.565, 0.656, 0.665, 0.897]
```

Schemat Blokowy:

```
[Start]
   ↓
[Find min and max values]
   ↓
[Initialize buckets]
   ↓
[Distribute elements into buckets]
   ↓
[Sort each bucket]
   ↓
[Concatenate all sorted buckets]
   ↓
[Stop]
```

Zalety:

- Szybki dla danych równomiernie rozłożonych

- Stabilny
- Możliwość równoległego sortowania kubełków

Wady:

- Wymaga dodatkowej pamięci na kubełki
- Niewydajny dla danych nierównomiernie rozłożonych

Zastosowania:

- Sortowanie liczb zmiennoprzecinkowych
- Implementacje sortowania w rozproszonych systemach

Algorytmy Grafowe

Algorytmy grafowe to zestaw metod służących do przetwarzania grafów w celu rozwiązania różnych problemów, takich jak znajdowanie najkrótszych ścieżek, wykrywanie cykli, czy znajdowanie minimalnych drzew rozpinających.

Przeszukiwanie Wszerz (BFS)

Opis: Przeszukiwanie wszerz (ang. Breadth-First Search - BFS) to algorytm przeszukiwania grafu, który eksploruje wszystkie węzły na danym poziomie przed przejściem do kolejnego poziomu.

Zastosowania:

- Znajdowanie najkrótszej ścieżki w grafie nieskierowanym lub skierowanym
- Sprawdzanie spójności grafu
- Znajdowanie komponentów spójności

Implementacja:

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")
        for neighbor, _ in graph.adj_list[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Przykład Użycia:

```python
g = GraphAdjacencyList(directed=False)
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'D')
g.add_edge('B', 'E')
g.add_edge('C', 'F')

print("BFS Traversal from A:")
bfs(g, 'A')  # Wyjście: A B C D E F
```

Złożoność:

- **Czas:** O(V + E), gdzie V to liczba wierzchołków, a E liczba krawędzi
- **Pamięć:** O(V)

Schemat Blokowy:

```
[Start]
   ↓
[Initialize queue with start node]
   ↓
[Initialize visited set with start node]
   ↓
[While queue is not empty]
       ↓
    [Dequeue vertex from queue]
       ↓
    [Visit vertex]
       ↓
    [Enqueue unvisited neighbors]
   ↓
[End While]
   ↓
[Stop]
```

Przeszukiwanie Wgłąb (DFS)

Opis: Przeszukiwanie wgłąb (ang. Depth-First Search - DFS) to algorytm przeszukiwania grafu, który eksploruje jak najgłębiej wzdłuż każdego gałęzi przed cofnięciem się.

Zastosowania:

- Sprawdzanie cykli w grafie
- Topologiczne sortowanie
- Znajdowanie komponentów silnie spójnych
- Rozwiązywanie problemów typu labirynt

Implementacja Rekurencyjna:

```python
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbor, _ in graph.adj_list[start]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)
```

Implementacja Iteracyjna:

```python
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            # Dodawanie sąsiadów do stosu w odwrotnej kolejności, aby zachować
kolejność odwiedzeń
            for neighbor, _ in reversed(graph.adj_list[vertex]):
                if neighbor not in visited:
                    stack.append(neighbor)
```

Przykład Użycia:

```python
g = GraphAdjacencyList(directed=False)
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'D')
g.add_edge('B', 'E')
g.add_edge('C', 'F')

print("DFS Recursive Traversal from A:")
dfs_recursive(g, 'A')  # Wyjście: A B D E C F

print("\nDFS Iterative Traversal from A:")
dfs_iterative(g, 'A')  # Wyjście: A C F B E D
```

Złożoność:

- **Czas:** O(V + E)
- **Pamięć:** O(V)

Schemat Blokowy (Rekurencyjny):

```
[Start]
   ↓
[Mark current node as visited]
   ↓
[Process current node]
   ↓
[Recurse for each unvisited neighbor]
   ↓
[Stop]
```

Algorytm Dijkstry

Opis: Algorytm Dijkstry znajduje najkrótsze ścieżki z jednego wierzchołka (źródła) do wszystkich innych wierzchołków w grafie ważonym, bez krawędzi o ujemnych wagach.

Zastosowania:

- Nawigacja GPS
- Najkrótsza ścieżka w sieciach komputerowych
- Analiza sieci transportowych

Implementacja:

```python
import heapq

def dijkstra(graph, start):
    distances = {vertex: float('infinity') for vertex in graph.adj_list}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue  # Ignoruj starsze wpisy

        for neighbor, weight in graph.adj_list[current_vertex]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

Przykład Użycia:

```python
g = GraphAdjacencyList(directed=False)
g.add_edge('A', 'B', 1)
```

```
g.add_edge('A', 'C', 4)
g.add_edge('B', 'C', 2)
g.add_edge('B', 'D', 5)
g.add_edge('C', 'D', 1)

distances = dijkstra(g, 'A')
print("Shortest distances from A:", distances)
# Wyjście: Shortest distances from A: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

Złożoność:

- **Czas:** O(E + V log V) przy użyciu kolejki priorytetowej
- **Pamięć:** O(V)

Schemat Blokowy:

```
[Start]
   ↓
[Initialize distances to infinity]
   ↓
[Set distance of start node to 0]
   ↓
[Initialize priority queue with start node]
   ↓
[While priority queue is not empty]
       ↓
    [Extract node with smallest distance]
       ↓
    [For each neighbor]
           ↓
        [Calculate new distance]
           ↓
        [If new distance is smaller, update and enqueue]
   ↓
[End While]
   ↓
[Stop]
```

Algorytm A*

Opis: Algorytm A* jest rozszerzeniem algorytmu Dijkstry, który używa funkcji heurystycznej do kierowania przeszukiwaniem w stronę celu, co często prowadzi do szybszego znalezienia najkrótszej ścieżki.

Zastosowania:

- Nawigacja GPS
- Gry komputerowe (ścieżkowanie postaci)
- Robotyka

Implementacja:

```python
import heapq

def heuristic(a, b):
    # Przykładowa heurystyka: odległość Manhattan dla siatki
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)

def a_star(graph, start, goal):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {vertex: float('infinity') for vertex in graph.adj_list}
    g_score[start] = 0
    f_score = {vertex: float('infinity') for vertex in graph.adj_list}
    f_score[start] = heuristic(start, goal)

    while open_set:
        current = heapq.heappop(open_set)[1]

        if current == goal:
            # Odtworzenie ścieżki
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for neighbor, weight in graph.adj_list[current]:
            tentative_g_score = g_score[current] + weight
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic(neighbor, goal)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return None  # Ścieżka nie istnieje
```

Przykład Użycia:

```python
g = GraphAdjacencyList(directed=False)
g.add_edge('A', 'B', 1)
g.add_edge('A', 'C', 4)
g.add_edge('B', 'C', 2)
g.add_edge('B', 'D', 5)
g.add_edge('C', 'D', 1)

path = a_star(g, 'A', 'D')
```

```
print("Path from A to D:", path)  # Wyjście: Path from A to D: ['A', 'B', 'C',
'D']
```

Złożoność:

- **Czas:** O(E), zależnie od funkcji heurystycznej
- **Pamięć:** O(V)

Schemat Blokowy:

```
[Start]
   ↓
[Initialize open set with start node]
   ↓
[Initialize g_score and f_score]
   ↓
[While open set is not empty]
       ↓
    [Extract node with smallest f_score]
       ↓
    [If node is goal, reconstruct path]
       ↓
    [For each neighbor]
           ↓
        [Calculate tentative_g_score]
           ↓
        [If tentative_g_score is better, update scores and enqueue]
   ↓
[End While]
   ↓
[Stop]
```

Algorytm Kruskala

Opis: Algorytm Kruskala znajduje minimalne drzewo rozpinające (MST) w grafie, wybierając krawędzie w kolejności rosnącej wagi i łącząc wierzchołki, unikając cykli.

Zastosowania:

- Minimalne drzewo rozpinające w sieciach komputerowych
- Projektowanie sieci telekomunikacyjnych
- Problemy optymalizacji

Implementacja:

```
class DisjointSet:
    def __init__(self, vertices):
        self.parent = {vertex: vertex for vertex in vertices}
```

```
    def find(self, vertex):
        if self.parent[vertex] != vertex:
            self.parent[vertex] = self.find(self.parent[vertex])  # Path
compression
        return self.parent[vertex]

    def union(self, vertex1, vertex2):
        root1 = self.find(vertex1)
        root2 = self.find(vertex2)
        if root1 != root2:
            self.parent[root2] = root1

def kruskal(graph):
    result = []  # MST
    i, e = 0, 0  # Indeksy
    # Sortowanie wszystkich krawędzi według wagi
    edges = []
    for src in graph.adj_list:
        for dest, weight in graph.adj_list[src]:
            edges.append((weight, src, dest))
    edges = sorted(edges)

    ds = DisjointSet(graph.adj_list.keys())

    while e < len(graph.adj_list) - 1 and i < len(edges):
        weight, src, dest = edges[i]
        i += 1
        root1 = ds.find(src)
        root2 = ds.find(dest)

        if root1 != root2:
            e += 1
            result.append((src, dest, weight))
            ds.union(root1, root2)

    return result
```

Przykład Użycia:

```
g = GraphAdjacencyList(directed=False)
g.add_edge('A', 'B', 1)
g.add_edge('A', 'C', 3)
g.add_edge('B', 'C', 2)
g.add_edge('B', 'D', 4)
g.add_edge('C', 'D', 5)

mst = kruskal(g)
print("Edges in MST:")
for edge in mst:
    print(edge)
# Wyjście:
# ('A', 'B', 1)
```

```
# ('B', 'C', 2)
# ('B', 'D', 4)
```

Złożoność:

- **Czas:** O(E log E) lub O(E log V)
- **Pamięć:** O(V + E)

Schemat Blokowy:

```
[Start]
   ↓
[Sort all edges by increasing weight]
   ↓
[Initialize disjoint sets for each vertex]
   ↓
[Initialize MST as empty]
   ↓
[For each edge in sorted order]
       ↓
    [If adding edge does not form a cycle]
       ↓
    [Add edge to MST]
       ↓
    [Union the sets]
   ↓
[End For]
   ↓
[Return MST]
   ↓
[Stop]
```

Algorytm Prima

Opis: Algorytm Prima również znajduje minimalne drzewo rozpinające (MST) w grafie. Wybiera dowolny węzeł jako początkowy i iteracyjnie dodaje najtańszą krawędź, która łączy węzeł z drzewem z węzłem spoza drzewa.

Zastosowania:

- Minimalne drzewo rozpinające w sieciach komputerowych
- Projektowanie sieci telekomunikacyjnych
- Problemy optymalizacji

Implementacja:

```
import heapq

def prim(graph, start):
    mst = []
```

```
    visited = set()
    min_heap = [(0, start, None)]  # (waga, wierzchołek, rodzic)

    while min_heap and len(mst) < len(graph.adj_list) - 1:
        weight, vertex, parent = heapq.heappop(min_heap)
        if vertex not in visited:
            visited.add(vertex)
            if parent is not None:
                mst.append((parent, vertex, weight))
            for neighbor, w in graph.adj_list[vertex]:
                if neighbor not in visited:
                    heapq.heappush(min_heap, (w, neighbor, vertex))

    return mst
```

Przykład Użycia:

```
g = GraphAdjacencyList(directed=False)
g.add_edge('A', 'B', 1)
g.add_edge('A', 'C', 3)
g.add_edge('B', 'C', 2)
g.add_edge('B', 'D', 4)
g.add_edge('C', 'D', 5)

mst = prim(g, 'A')
print("Edges in MST:")
for edge in mst:
    print(edge)
# Wyjście:
# ('A', 'B', 1)
# ('B', 'C', 2)
# ('B', 'D', 4)
```

Złożoność:

- **Czas:** O(E log V)
- **Pamięć:** O(V + E)

Schemat Blokowy:

```
[Start]
   ↓
[Initialize MST as empty]
   ↓
[Initialize min_heap with start node]
   ↓
[Initialize visited set]
   ↓
[While min_heap is not empty and MST is incomplete]
```

```
        ↓
     [Extract edge with minimum weight]
        ↓
     [If vertex not visited]
            ↓
         [Add edge to MST]
            ↓
         [Mark vertex as visited]
            ↓
         [Add all adjacent edges to min_heap]
    ↓
 [End While]
    ↓
 [Return MST]
    ↓
 [Stop]
```

Rekurencja i Dynamiczne Przydzielanie Pamięci

Przykłady Rekurencji

Opis: Rekurencja to technika, w której funkcja wywołuje samą siebie w celu rozwiązania problemu. Rekurencja jest szczególnie użyteczna w problemach podziału na podproblemy, takich jak przeszukiwanie drzew, sortowanie, czy rozwiązywanie problemów kombinatorycznych.

Przykłady:

1. **Obliczanie Silni**
2. **Rozwiązywanie Problemów Wież Hanoi**
3. **Algorytm Quick Sort**
4. **Przeszukiwanie Drzewa**

Przykład Implementacji (Python - Silnia):

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Wyjście: 120
```

Przykład Implementacji (Python - Wieże Hanoi):

```python
def hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Przenieś dysk 1 z {source} na {target}")
        return
    hanoi(n - 1, source, auxiliary, target)
```

```
    print(f"Przenieś dysk {n} z {source} na {target}")
    hanoi(n - 1, auxiliary, target, source)

hanoi(3, 'A', 'C', 'B')
# Wyjście:
# Przenieś dysk 1 z A na C
# Przenieś dysk 2 z A na B
# Przenieś dysk 1 z C na B
# Przenieś dysk 3 z A na C
# Przenieś dysk 1 z B na A
# Przenieś dysk 2 z B na C
# Przenieś dysk 1 z A na C
```

Przykład Implementacji (Python - Przeszukiwanie DFS):

```
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbor, _ in graph.adj_list[start]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)
```

Techniki Rekurencyjne

Rekurencja Ogonowa (Tail Recursion)

Opis: Rekurencja ogonowa to forma rekurencji, gdzie wywołanie rekurencyjne jest ostatnią operacją w funkcji. Pozwala to na optymalizację przez kompilator (Tail Call Optimization - TCO), eliminując dodatkowe stosy wywołań.

Przykład (Python nie wspiera TCO, ale struktura jest pokazana):

```
def tail_recursive_factorial(n, acc=1):
    if n == 0:
        return acc
    return tail_recursive_factorial(n - 1, acc * n)

print(tail_recursive_factorial(5))  # Wyjście: 120
```

Memoizacja

Opis: Memoizacja to technika optymalizacji, która polega na przechowywaniu wyników kosztownych wywołań funkcji i zwracaniu zapisanych wyników, gdy te same wywołania wystąpią ponownie.

Przykład Implementacji (Python - Fibonacci z Memoizacją):

```python
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]

print(fibonacci(10))  # Wyjście: 55
```

Zalety:

- Znaczne zmniejszenie liczby wywołań rekurencyjnych
- Poprawa wydajności dla problemów z powtarzającymi się podproblemami

Wady:

- Zwiększone zużycie pamięci
- Konieczność zarządzania pamięcią cache

Dynamiczne Przydzielanie Pamięci

Opis: Dynamiczne przydzielanie pamięci to technika zarządzania pamięcią, która umożliwia przydzielanie i zwalnianie pamięci w trakcie działania programu. Jest to kluczowe dla tworzenia dynamicznych struktur danych, takich jak listy, stosy, kolejki czy grafy.

Zastosowania:

- Tworzenie dynamicznych struktur danych
- Optymalizacja wykorzystania pamięci
- Zarządzanie pamięcią w aplikacjach o zmiennym rozmiarze danych

Przykład Implementacji (Python - Dynamiczna Lista):

```python
dynamic_list = []
for i in range(10):
    dynamic_list.append(i)
print(dynamic_list)  # Wyjście: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Zalety:

- Elastyczność w zarządzaniu pamięcią
- Efektywne wykorzystanie pamięci dla zmiennych rozmiarów danych

Wady:

- Zarządzanie pamięcią może być skomplikowane
- Potencjalne wycieki pamięci, jeśli pamięć nie jest odpowiednio zwalniana

Przykład Implementacji (C++ - Dynamiczna Tablica):

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

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
    int* arr = new int[n];  // Dynamiczne przydzielanie pamięci

    for(int i = 0; i < n; i++) {
        arr[i] = i * 2;
    }

    cout << "Array elements: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr;  // Zwolnienie pamięci
    return 0;
}
```

Schemat Blokowy:

```
[Start]
   ↓
[Request memory allocation]
   ↓
[Use allocated memory]
   ↓
[Free memory when done]
   ↓
[Stop]
```

Zalety:

- Optymalne wykorzystanie pamięci
- Możliwość tworzenia dynamicznych struktur danych

Wady:

- Potencjalne problemy z zarządzaniem pamięcią (wycieki, fragmentacja)
- Zwiększona złożoność programowania

Analiza Złożoności Algorytmów

Opis: Analiza złożoności algorytmów to ocena wydajności algorytmu w zależności od wielkości danych wejściowych. Złożoność może być mierzona pod względem czasu wykonania (czasowa) lub zużycia pamięci (przestrzenna).

Notacja Big O

Opis: Notacja Big O opisuje górną granicę złożoności czasowej algorytmu w zależności od wielkości wejścia. Umożliwia porównanie wydajności różnych algorytmów niezależnie od sprzętu.

Popularne Notacje:

- **O(1):** Stała złożoność
- **O(log n):** Logarytmiczna złożoność
- **O(n):** Liniowa złożoność
- **O(n log n):** Liniowo-logarytmiczna złożoność
- **O(n^2):** Kwadratowa złożoność
- **O(2^n):** Eksponencjalna złożoność
- **O(n!):** Silnia złożoność

Przykłady:

- **O(1):** Dostęp do elementu w tablicy
- **O(log n):** Przeszukiwanie binarne
- **O(n):** Przeszukiwanie liniowe
- **O(n log n):** Sortowanie szybkie, sortowanie scalanie
- **O(n^2):** Sortowanie bąbelkowe, sortowanie przez wybór

Tabela Złożoności:

Algorytm	Najlepszy	Średni	Najgorszy
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Quick Sort	O(n log n)	O(n log n)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)
Radix Sort	O(nk)	O(nk)	O(nk)
Counting Sort	O(n + k)	O(n + k)	O(n + k)
Binary Search	O(1)	O(log n)	O(log n)
Linear Search	O(1)	O(n)	O(n)

Notacja Big Ω i Big Θ

Notacja Big Ω:

- Opisuje dolną granicę złożoności algorytmu, czyli minimalną liczbę operacji, które algorytm wykona dla danego rozmiaru wejścia.

Notacja Big Θ:

- Opisuje dokładną złożoność algorytmu, określając zarówno górną, jak i dolną granicę złożoności.

Przykład:

- Algorytm sortowania bąbelkowego ma złożoność Big Ω(n) (najlepszy przypadek) oraz Big $O(n^2)$.

Przykłady Analizy

Przykład 1: Bubble Sort

- **Opis:** Dwa zagnieżdżone pętle, każda iterująca n razy.
- **Złożoność:** $O(n^2)$
- **Analiza:**
  - Pierwsza pętla: O(n)
  - Druga pętla: O(n) w każdym kroku pierwszej pętli
  - Razem: $O(n) * O(n) = O(n^2)$

Przykład 2: Binary Search

- **Opis:** Dzielenie zakresu wyszukiwania na pół w każdym kroku.
- **Złożoność:** O(log n)
- **Analiza:**
  - W każdym kroku liczba możliwych elementów do przeszukania jest dzielona przez 2
  - Liczba kroków: $\log_2(n)$
  - Złożoność: O(log n)

Przykład 3: Merge Sort

- **Opis:** Dzielenie listy na pół log n razy i scalanie każdej części w czasie liniowym.
- **Złożoność:** O(n log n)
- **Analiza:**
  - Dzielenie: O(log n) poziomów podziału
  - Scalanie na każdym poziomie: O(n)
  - Razem: O(n log n)

Analiza Przestrzenna

Opis: Analiza przestrzenna ocenia ilość pamięci potrzebnej przez algorytm w zależności od rozmiaru danych wejściowych.

Przykłady:

- **O(1):** Stała ilość pamięci (np. algorytm sortowania bąbelkowego, który sortuje w miejscu)
- **O(n):** Pamięć liniowa (np. sortowanie scalanie, które wymaga dodatkowej tablicy)

Przykład:

- Algorytm Quick Sort sortuje w miejscu, więc jego złożoność przestrzenna to O(log n) (stos rekurencji).
- Algorytm Merge Sort wymaga dodatkowej pamięci O(n).

Podsumowanie

Opanowanie algorytmów i struktur danych jest niezbędne dla każdego informatyka. Wiedza ta pozwala na tworzenie efektywnych i skalowalnych rozwiązań, które są fundamentem nowoczesnych technologii. Zrozumienie różnych struktur danych oraz umiejętność wyboru odpowiedniego algorytmu do konkretnego problemu jest kluczowa dla optymalizacji aplikacji i systemów informatycznych.

Kluczowe Punkty:

- **Struktury Danych:** Wybór odpowiedniej struktury danych jest podstawą dla efektywnego algorytmu.
- **Algorytmy Sortowania:** Różne algorytmy sortowania mają swoje zalety i wady, zależnie od kontekstu użycia.
- **Algorytmy Grafowe:** BFS i DFS są podstawowymi algorytmami wykorzystywanymi w wielu zastosowaniach grafowych.
- **Rekurencja:** Potężne narzędzie do rozwiązywania problemów, ale wymaga ostrożności ze względu na zużycie pamięci.
- **Analiza Złożoności:** Pozwala na ocenę wydajności algorytmów i wybór optymalnych rozwiązań.