

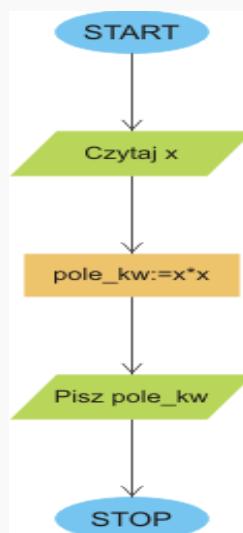
Klasyfikacja algorytmów

Klasyfikacja algorytmów ze względu na sposób konstruowania algorytmu

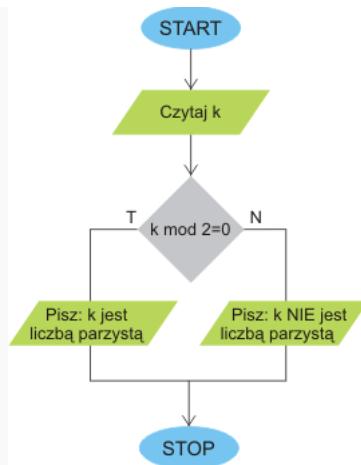
- **Dziel i zwyciężaj** — problem, który należy rozwiązać, jest dzielony na kilka mniejszych, a te znowu są dzielone aż do uzyskania problemów łatwych do rozwiązania. Algorytmy z tej grupy należą do najsukceszniejszych metod rozwiązywania problemów
- **Programowanie dynamiczne** — metoda podobna do metody dziel i zwyciężaj. Problem, który należy rozwiązać, jest dzielony na kilka mniejszych. Wyniki analizy cząstkowych problemów wykorzystuje się do rozwiązania głównego problemu.
- **Metoda zachłanna** — nie jest przeprowadzana dokładna analiza problemu, tylko wybierane jest rozwiązanie, które w danym momencie wydaje się najkorzystniejsze.
- **Poszukiwanie i wyliczanie** — przeszukiwany jest zbiór danych aż do znalezienia rozwiązania.
- **Heurystyka** — na podstawie niepełnych danych tworzony jest algorytm, który działa w sposób najbardziej prawdopodobny. Metody heurystyczne nie zapewniają otrzymania poprawnego rozwiązania; powstałe algorytmy dają zawsze rozwiązania przybliżone.

Klasyfikacja algorytmów ze względu na kolejność wykonywania działań

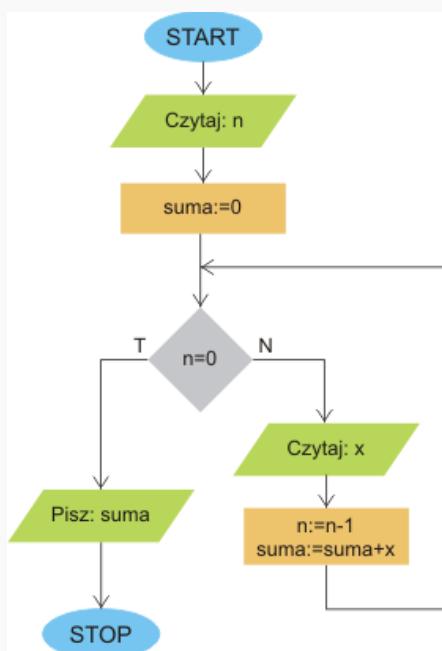
- **Liniowy** — kolejne kroki w algorytmie są wykonywane w kolejności, w jakiej zostały zapisane. Żaden krok nie może być pominięty ani powtórzony



- **Warunkowy (z rozgałęzieniem)** — wykonanie zależy od spełnienia lub niespełnienia określonego warunku.



- **Z pętlą (cykliczny)** — grupa poleceń jest powtarzana wielokrotnie. Liczba powtórzeń może być z góry określona lub grupa poleceń jest powtarzana aż do spełnienia określonego warunku.



Klasyfikacja algorytmów ze względu na sposób wykonywania operacji

- **Sekwencyjne** — operacje w algorytmie są wykonywane w kolejności, w jakiej zostały opisane.
- **Iteracyjne** — niektóre kroki są powtarzane aż do spełnienia wymaganego warunku.
- **Rekurencyjne** — tworzona jest formuła powtarzająca dane i odwołującą się do niej samej.
Zakończenie wywoływanego formuły następuje po spełnieniu warunku zakończenia rekurencji.

Klasyfikacja algorytmów ze względu na obszar zastosowań

- **Matematyczne** — wykonują obliczenia numeryczne
- **Przeszukujące** — przeszukują zbiór danych w celu znalezienia określonego elementu
- **Porządkujące** — ustawiają elementy zbioru w określonej kolejności
- **Rekurencyjne** — rozwiązuje problemy, które po podzieleniu na mniejsze części stanowią kopię głównego problemu
- **Szyfrujące** — kodują dane w ten sposób, że ich odczyt jest niemożliwy bez znajomości klucza kodującego

Złożoność obliczeniowa algorytmu

Złożoność obliczeniowa algorytmu określa ilość zasobów (pamięci, czasu) niezbędnych do rozwiązania problemu.

- **Złożoność czasowa** algorytmu jest rozpatrywana jako ilość czasu potrzebnego do rozwiązania problemu w zależności od liczby danych wejściowych. Podawanie złożoności obliczeniowej w jednostkach czasu jest jednak nieprecyzyjne, bo wynik zależy również od szybkości działania komputera. Dlatego złożoność obliczeniowa algorytmu najczęściej podawana jest w liczbie wykonanych operacji i jest to największa liczba operacji dominujących wykonywanych w algorytmie dla dowolnego układu danych.
- **Złożoność pamięciowa** algorytmu określa wielkość pamięci operacyjnej komputera, która jest potrzebna do przechowywania danych wejściowych, danych pośrednich oraz ostatecznych wyników obliczeń.

Złożoność czasowa oraz złożoność pamięciowa algorytmu często zależą od **postaci przetwarzanych danych**, dlatego można je przedstawić jako złożoność:

- **pesymistyczną** — określa zużycie zasobów dla najgorszego przypadku,
- **oczekiwana** — określa zużycie zasobów dla uśrednionych wszystkich możliwych przypadków lub dla typowych przypadków,
- **optimistyczną** — określa zużycie zasobów dla najkorzystniejszego przypadku.

Jaki język najlepiej implementować algorytmy?

- **Python**: Najlepszy do nauki i prototypowania. Jest czytelny, ma wbudowane struktury danych (listy, słowniki), biblioteki jak NumPy do zaawansowanych obliczeń. Idealny dla początkujących, edukacji i szybkiego testowania. Wadą jest wolniejsza wydajność w porównaniu do języków komplikowanych.
- **C++ lub Java**: Najlepsze do konkursów programistycznych (np. ACM ICPC, Google Code Jam) i aplikacji wymagających wysokiej wydajności. C++ jest szybki, ale bardziej skomplikowany w składni.
- **Inne opcje**: JavaScript do webowych aplikacji, Rust do bezpiecznych systemów. Ogólnie, zacznij od Pythona, a potem przejdź na C++ jeśli potrzebujesz prędkości.

Jeśli implementujesz algorytmy do nauki, użyj Pythona. Do produkcji – zależy od kontekstu (np. ML: Python z TensorFlow; gry: C++) .

Kategorie algorytmów z przykładami 1. Algorytmy sortowania (Sorting Algorithms)
Te algorytmy porządkują elementy w liście. Są kluczowe w przetwarzaniu danych.

- **Bubble Sort**: Prosty, ale nieefektywny ($O(n^2)$). Najlepszy do małych list lub edukacji – porównuje sąsiednie elementy i zamienia je.

```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        for j in range(0, n-i-1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]  
    return arr
```

```
# Przykład użycia
lista = [64, 34, 25, 12, 22, 11, 90]
print(bubble_sort(lista)) # Wyjście: [11, 12, 22, 25, 34, 64, 90]
```

- **Quick Sort:** Efektywny (średnio $O(n \log n)$). Najlepszy do dużych zbiorów danych, gdy potrzebna jest szybkość (używa divide-and-conquer).

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    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
lista = [64, 34, 25, 12, 22, 11, 90]
print(quick_sort(lista)) # Wyjście: [11, 12, 22, 25, 34, 64, 90]
```

- **Merge Sort:** Stabilny i efektywny ($O(n \log n)$). Najlepszy do sortowania dużych list, gdzie stabilność (zachowanie kolejności równych elementów) jest ważna, np. w bazach danych.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# Przykład użycia
lista = [64, 34, 25, 12, 22, 11, 90]
print(merge_sort(lista)) # Wyjście: [11, 12, 22, 25, 34, 64, 90]
```

2. Algorytmy wyszukiwania (Searching Algorithms)

Służą do znajdowania elementów w strukturach danych.

- **Linear Search:** Prosty ($O(n)$). Najlepszy do małych, nieposortowanych list – sprawdza po kolej.

```
def linear_search(arr, target):  
    for i in range(len(arr)):  
        if arr[i] == target:  
            return i  
    return -1  
  
# Przykład użycia  
lista = [64, 34, 25, 12, 22, 11, 90]  
print(linear_search(lista, 22)) # Wyjście: 4
```

- **Binary Search:** Efektywny ($O(\log n)$). Najlepszy do posortowanych list/tablic – dzieli zakres na pół.

```
def binary_search(arr, target):  
    low, high = 0, len(arr) - 1  
    while low <= high:  
        mid = (low + high) // 2  
        if arr[mid] == target:  
            return mid  
        elif arr[mid] < target:  
            low = mid + 1  
        else:  
            high = mid - 1  
    return -1
```

```
# Przykład użycia (lista musi być posortowana)  
lista = [11, 12, 22, 25, 34, 64, 90]  
print(binary_search(lista, 25)) # Wyjście: 3
```

3. Algorytmy na grafach (Graph Algorithms)

Do przetwarzania struktur jak sieci społeczne czy mapy.

- **BFS (Breadth-First Search):** $O(n + m)$. Najlepszy do znajdowania najkrótszej ścieżki w niesważonym grafie, np. w labiryntach.

```
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[vertex]:  
            if neighbor not in visited:  
                visited.add(neighbor)  
                queue.append(neighbor)
```

```
# Przykład użycia (graf jako słownik)
graph = {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []}
bfs(graph, 'A') # Wyjście: A B C D E F
```

- **DFS (Depth-First Search):** $O(n + m)$. Najlepszy do wykrywania cykli lub eksploracji głębszej, np. w rozwiązywaniu zagadek.

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

```
# Przykład użycia
graph = {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []}
dfs(graph, 'A') # Wyjście: A B D E F C
```

- **Dijkstra's Algorithm:** $O((n + m) \log n)$. Najlepszy do najkrótszej ścieżki w ważonym grafie z dodatnimi wagami, np. nawigacja GPS.

```
import heapq
```

```
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]
    while pq:
        current_distance, current_node = heapq.heappop(pq)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return distances
```

```
# Przykład użycia (graf z wagami)
graph = {'A': {'B': 1, 'C': 4}, 'B': {'C': 2, 'D': 5}, 'C': {'D': 1}, 'D': {}}
print(dijkstra(graph, 'A')) # Wyjście: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

4. Algorytmy dynamicznego programowania (Dynamic Programming) Rozwiązuje problemy przez zapamiętywanie podproblemów.

- **Fibonacci (z memoizacją):** $O(n)$. Najlepszy do sekwencji rekurencyjnych, np. modelowanie wzrostu populacji.

```
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]
```

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

- **0/1 Knapsack:** $O(nW)$. Najlepszy do optymalizacji zasobów, np. pakowanie plecaka z maksymalną wartością.

```
def knapsack(weights, values, capacity):
    n = len(values)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]
```

```
# Przykład użycia
weights = [1, 3, 4]
values = [1, 4, 5]
capacity = 7
print(knapsack(weights, values, capacity)) # Wyjście: 9
```

5. Inne ważne algorytmy

- **Greedy Algorithm (np. Huffman Coding):** Najlepszy do optymalizacji lokalnej, np. kompresja danych. Nie zawsze optymalny globalnie.
- **Divide and Conquer (np. w Quick Sort):** Najlepszy do dużych problemów, dzieląc je na mniejsze.
- **Backtracking (np. Sudoku Solver):** Najlepszy do eksploracji wszystkich możliwości, np. gry logiczne.