

# Metody przeszukiwania wyczerpującego

Autor: Filip Lisowski Data: 14.10.2025 Przedmiot: Algorytmy i struktury danych

## 🔗 Zadanie 1 - Złamanie hasła alfanumerycznego.

**Opis problemu:** Stwórz funkcję `zlam_haslo(prawdziwe_haslo)`. Ma na ślepo sprawdzać wszystkie 3-znakowe kombinacje ze znaków (a-z i 0-9), aż trafi na `prawdziwe_haslo`.

**Przykład działania:**

Znaleziono hasło "a01" po 37 próbach.  
Znaleziono hasło "z99" po 46656 próbach.

**Kod programu:**

```
def zlam_haslo(prawdziwe_haslo: str) -> int:
    alfabet = "abcdefghijklmnopqrstuvwxyz0123456789"
    proby = 0

    for a in alfabet:
        for b in alfabet:
            for c in alfabet:
                proby += 1
                haslo = a + b + c
                if haslo == prawdziwe_haslo:
                    print(f'Znaleziono hasło "{haslo}" po {proby} próbach.')
                    return proby

    print("Hasło nie należy do przestrzeni przeszukiwania.")
    return proby

zlam_haslo("999")
```

**Jaka jest złożoność obliczeniowa tego algorytmu?**

- Czasowa:  $O(36^3)$
- Pamięciowa:  $O(1)$

**Ile wynosi maksymalna liczba prób, które program musi wykonać w najgorszym przypadku?**

- Maksymalna liczba prób:  $36^3 = 46656$

## 🔗 Zadanie 2 - Znajdowanie trójki liczb.

**Opis problemu:** Stwórz funkcję `znajdz_trojke(lista, target)`, która ma zwrócić **trzy różne indeksy** ( $i$ ,  $j$ ,  $k$ ), takie, że  $lista[i] + lista[j] + lista[k] == target$ .

Wystarczy **pierwsza** pasująca trójka; jak nie ma - zwróć `None`.

### Przykład działania:

```
znajdz_trojke([1, 5, 2, 8, 4], 11)      # Zwraca (0, 2, 3) -> bo 1 + 2 + 8 = 11
znajdz_trojke([10, 20, 3, 4, 5], 12)    # Zwraca (2, 3, 4) -> bo 3 + 4 + 5 = 12
znajdz_trojke([1, 2, 3, 4], 20)         # Zwraca None
znajdz_trojke([-1, 0, 1, 2, -1, -4], 0) # Zwraca (0, 1, 2) -> bo -1 + 0 + 1 = 0
```

### Kod programu:

```
def znajdz_trojke(lista, target):
    n = len(lista)
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if lista[i] + lista[j] + lista[k] == target:
                    return i, j, k
    return None

print(znajdz_trojke([1, 5, 2, 8, 4], 11))
print(znajdz_trojke([1, 2, 3, 4], 20))
```

### Jaka jest złożoność obliczeniowa tego algorytmu?

- Czasowa:  $O(n^3)$
- Pamięciowa:  $O(1)$

### Jak zmieniłaby się złożoność, gdybyśmy szukali czwórki liczb?

- Dla czwórki liczb:  $O(n^4)$

## 🔗 Zadanie 3: Problem sumy podzbioru (Subset Sum)

Masz **unikalne, dodatnie** liczby i `target`. Funkcja `znajdz_podzbior_sumy(liczby, target)` ma zwrócić podzbiór (lista elementów), który sumuje się **dokładnie** do `target`.

Jeśli nic nie pasuje - zwróć [] lub None.

### Przykład działania:

```
znajdz_trojke([1, 5, 2, 8, 4], 11)    # Zwraca (0, 2, 3) -> bo 1 + 2 + 8 = 11
znajdz_trojke([10, 20, 3, 4, 5], 12)  # Zwraca (2, 3, 4) -> bo 3 + 4 + 5 = 12
znajdz_trojke([1, 2, 3, 4], 20)       # Zwraca None
znajdz_trojke([-1, 0, 1, 2, -1, -4], 0) # Zwraca (0, 1, 2) -> bo -1 + 0 + 1 = 0
```

### Kod programu:

```
def znajdz_podzbior_sumy(liczby, target):
    n = len(liczby)

    for maska in range(1, 2**n):
        podzbior = []
        suma = 0
        for i in range(n):
            if maska & (1 << i):
                podzbior.append(liczby[i])
                suma += liczby[i]
        if suma == target:
            return podzbior
    return []

print(znajdz_podzbior_sumy([2, 5, 8, 3], 10))
```

### Jaka jest złożoność obliczeniowa tego algorytmu?

- Czasowa:  $O(n * 2^n)$
- Pamięciowa:  $O(n)$

---

## Zadanie 4 - Problem Komiwojażera w małej skali

Miasto startowe to **A** (indeks 0). Trzeba odwiedzić pozostałe **B, C, D** raz i wrócić do **A**, tak żeby trasa była **najkrótsza**. Funkcja ma zwrócić **trasę** i jej **długość**.

### Przykład danych wejściowych (macierz odległości):

```
# Miasta: A, B, C, D
dist = [
    [0, 10, 15, 20], # A
    [10, 0, 35, 25], # B
    [15, 35, 0, 30], # C
    [20, 25, 30, 0]  # D
]
```

### Przykład działania:

Sprawdzane trasy:

```
A -> B -> C -> D -> A = 95
A -> B -> D -> C -> A = 80
A -> C -> B -> D -> A = 95
A -> C -> D -> B -> A = 80
A -> D -> B -> C -> A = 95
A -> D -> C -> B -> A = 95
```

Najkrótsza trasa: A -> B -> D -> C -> A

Długość trasy: 80

### Kod programu:

```
def najkrotsza_trasa(macierz):
    najlepszy_koszt = float('inf')
    najlepsza_trasa = None

    for b in (1, 2, 3):
        for c in (1, 2, 3):
            if c == b:
                continue
            for d in (1, 2, 3):
                if d == b or d == c:
                    continue

                obecna_trasa = [0, b, c, d, 0]
                obecny_koszt = (
                    macierz[obecna_trasa[0]][obecna_trasa[1]] +
                    macierz[obecna_trasa[1]][obecna_trasa[2]] +
                    macierz[obecna_trasa[2]][obecna_trasa[3]] +
                    macierz[obecna_trasa[3]][obecna_trasa[4]]
                )

                if obecny_koszt < najlepszy_koszt:
                    najlepszy_koszt = obecny_koszt
```

```
        najlepsza_trasa = obecna_trasa

    return najlepsza_trasa, int(najlepszy_koszt)

macierz_odleglosci = [
    [0, 10, 15, 20], # A
    [10, 0, 35, 25], # B
    [15, 35, 0, 30], # C
    [20, 25, 30, 0]  # D
]

trasa_wynikowa, koszt_wynikowy = najkrotsza_trasa(macierz_odleglosci)

miasta = ["A", "B", "C", "D"]
trasa_literowa = [miasta[i] for i in trasa_wynikowa]

print("Najkrótsza trasa:", " -> ".join(trasa_literowa))
print("Długość trasy:", koszt_wynikowy)
```

### Jaka jest złożoność obliczeniowa tego algorytmu?

- Czasowa:  $O(n!)$
- Pamięciowa:  $O(n)$

---

### Podsumowanie:

Wszystkie powyższe zadania wykorzystują metodę **brute-force**. Dodatkowo wykonane zadania znajdują się na repozytorium [GitHub](#)