ALGORYTMY

ALGORYTMY

Tworzenie listy z 26 zerami

```
my_list = [0] * 26
print("Początkowa lista:", my_list)
```

Tworzenie listy z 100 False

```
my_list = [False] * 100
print("Początkowa lista:", my_list)
```

Dodawanie elementów do listy przy użyciu '+ [i]'

```
for i in range(26):
    my_list = myList + [i]  # Dodaje liczby od 0 do 25
print("Zaktualizowana lista:", my_list)
```

Dodawanie elementów do listy przy użyciu '+= [i]'

```
for i in range(26):
    my_list += [i+1]  # Dodaje liczby od 1 do 26
print("Zaktualizowana lista:", my_list)
```

Wypełnienie listy literami

```
letters = []
for i in range(26):
    letters.append(chr(ord('a') + i))
print(letters)
```

Zamiana liczby na listę cyfr

```
def rozbij_na_cyfry(liczba):
    lista_cyfr = []
    while liczba > 0:
        cyfra = liczba % 10
        lista_cyfr.append(cyfra)
        liczba = (liczba // 10)
    return lista_cyfr
```

```
def rozbij_na_cyfry(liczba):
    lista_cyfr = []
    for cyfra in str(liczba):
        lista_cyfr.append(int(cyfra))
    return lista_cyfr
```

Zamiana listy cyfr na liczbę

```
def lista_cyfr_na_liczbe(lista_cyfr):
    liczba = 0
    for cyfra in lista_cyfr:
        liczba = liczba * 10 + cyfra
    return liczba

#
# liczba = 0
# lista_cyfr = [1, 2, 3, 4]
# Pierwsza iteracja (cyfra = 1):
#
# liczba = liczba * 10 + cyfra
# liczba = 0 * 10 + 1 = 1
# Druga iteracja (cyfra = 2):
#
# liczba = liczba * 10 + cyfra
# liczba = 1 * 10 + 2 = 12
# Trzecia iteracja (cyfra = 3):
#
# liczba = liczba * 10 + cyfra
# liczba = 12 * 10 + 3 = 123
# Czwarta iteracja (cyfra = 4):
#
```

```
# liczba = liczba * 10 + cyfra
# liczba = 123 * 10 + 4 = 1234
# Zwracanie wyniku:
#
# Funkcja zwraca 1234.
```

```
def lista_cyfr_na_liczbe(lista_cyfr):
    liczba_list = []
    for cyfra in lista_cyfr:
        liczba_list = liczba_list + [str(cyfra)]
    liczba_str = "".join(liczba_list)
    return int(liczba_str)
```

Funkcja suma_cyfr

```
def suma_cyfr(n):
    pom = 0
    while n > 0:
        pom += n % 10  # Dodanie ostatniej cyfry do sumy
        n = n // 10  # Usunięcie ostatniej cyfry z liczby
    return pom

print(suma_cyfr(4))  # Wynik: 10
```

Suma liczb z przedziału

```
def suma_przedzial(a, b):
    s = 0
    for i in range(a, b + 1):  # Uwzględnienie końcowej wartości b
        s += i
    return s

print(suma_przedzial(3, 6))  # Wynik: 3 + 4 + 5 + 6 = 18
```

Zamiana liczb o dowolnej podstawie

```
def to_base(n, base):
    if n == 0:
        return "0"
    digits = []
    while n:
        digit = n % base
        digits.append(digit)
        n //= base
    digits.reverse()  # Odwrócenie listy

    result = ""
    for x in digits:
        result += str(x)  # Łączenie elementów w ciąg znaków
    return result

to_base(123232,7)
```

Oto szczegółowy krok po kroku proces zamiany liczby 27 na system binarny (podstawa 2) przy użyciu funkcji `to_base()`:

Krok po kroku proces:

Liczba: **27**
Podstawa: **2** (system binarny)

1. **Pierwszy krok (dzielenie 27 przez 2):**
   - 27 % 2 = **1** (reszta)
   - 27 // 2 = **13** (część całkowita)
   - Dodaj resztę (1) do listy cyfr.

   Cyfry: [1]
   Liczba do dalszego dzielenia: 13
2. **Drugi krok (dzielenie 13 przez 2):**
   - 13 % 2 = **1** (reszta)
   - 13 // 2 = **6** (część całkowita)
   - Dodaj resztę (1) do listy cyfr.

   Cyfry: [1, 1]
   Liczba do dalszego dzielenia: 6
3. **Trzeci krok (dzielenie 6 przez 2):**

- 6 % 2 = **0** (reszta)
- 6 // 2 = **3** (część całkowita)
- Dodaj resztę (0) do listy cyfr.

Cyfry: [1, 1, 0]
Liczba do dalszego dzielenia: 3

4. **Czwarty krok (dzielenie 3 przez 2):**

   - 3 % 2 = **1** (reszta)
   - 3 // 2 = **1** (część całkowita)
   - Dodaj resztę (1) do listy cyfr.

   Cyfry: [1, 1, 0, 1]
   Liczba do dalszego dzielenia: 1

5. **Piąty krok (dzielenie 1 przez 2):**

   - 1 % 2 = **1** (reszta)
   - 1 // 2 = **0** (część całkowita)
   - Dodaj resztę (1) do listy cyfr.

   Cyfry: [1, 1, 0, 1, 1]
   Liczba do dalszego dzielenia: 0 (koniec procesu)

Po zakończeniu dzielenia:

- Cyfry w odwrotnej kolejności to: **[1, 1, 0, 1, 1]**.
- Po odwróceniu listy otrzymujemy **[1, 1, 0, 1, 1]**, co odpowiada binarnej reprezentacji liczby **27**: **"11011"**.

Podsumowanie:

27 w systemie binarnym to 11011.

Sortowanie bąbelkowe

Funkcja `sortuj_babelkowo_lista` implementuje algorytm sortowania bąbelkowego (Bubble Sort). Jest to jeden z najprostszych algorytmów sortujących, który działa na zasadzie wielokrotnego porównywania sąsiednich elementów listy i ich zamiany, jeśli są w złej kolejności.
Proces ten powtarza się, aż lista będzie posortowana.

Krok po kroku:

1. Zmienna `n` przechowuje długość listy.
2. Pierwsza pętla `for i in range(n)` iteruje po elementach listy. Z każdą iteracją lista jest coraz bardziej uporządkowana.
3. Druga pętla `for j in range(0, n-i-1)` porównuje sąsiednie elementy. Dzięki `n-i-1` za każdym razem zakres porównań jest mniejszy, ponieważ na końcu listy po każdej iteracji znajdują się elementy w odpowiedniej kolejności.
4. Jeśli `lista[j] > lista[j + 1]`, następuje zamiana miejscami tych elementów.
5. Zwracana jest posortowana lista.

Przykład sortowania listy:

```python
def sortuj_babelkowo_lista(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):  # Mniejszy zakres, bo po każdej iteracji elementy na końcu są posortowane
            if lista[j] > lista[j + 1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista

# Przykład użycia:
my_list = [6, 3, 8, 5, 2, 7]
print("Przed sortowaniem:", my_list)
sorted_list = sortuj_babelkowo_lista(my_list)
print("Po sortowaniu:", sorted_list)
```

Kroki zamiany dla przykładu listy `[6, 3, 8, 5, 2, 7]`

Początkowa lista:

`[6, 3, 8, 5, 2, 7]`

1. Pierwsza iteracja (`i=0`):

- Porównanie: `6 > 3` → zamieniamy `6` i `3` → `[3, 6, 8, 5, 2, 7]`
- Porównanie: `6 < 8` → brak zamiany
- Porównanie: `8 > 5` → zamieniamy `8` i `5` → `[3, 6, 5, 8, 2, 7]`
- Porównanie: `8 > 2` → zamieniamy `8` i `2` → `[3, 6, 5, 2, 8, 7]`
- Porównanie: `8 > 7` → zamieniamy `8` i `7` → `[3, 6, 5, 2, 7, 8]`

Po pierwszej iteracji lista: `[3, 6, 5, 2, 7, 8]`

2. Druga iteracja (`i=1`):

- Porównanie: `3 < 6` → brak zamiany
- Porównanie: `6 > 5` → zamieniamy `6` i `5` → `[3, 5, 6, 2, 7, 8]`
- Porównanie: `6 > 2` → zamieniamy `6` i `2` → `[3, 5, 2, 6, 7, 8]`
- Porównanie: `6 < 7` → brak zamiany

Po drugiej iteracji lista: `[3, 5, 2, 6, 7, 8]`

3. Trzecia iteracja (`i=2`):

- Porównanie: `3 < 5` → brak zamiany
- Porównanie: `5 > 2` → zamieniamy `5` i `2` → `[3, 2, 5, 6, 7, 8]`
- Porównanie: `5 < 6` → brak zamiany

Po trzeciej iteracji lista: `[3, 2, 5, 6, 7, 8]`

4. Czwarta iteracja (`i=3`):

- Porównanie: `3 > 2` → zamieniamy `3` i `2` → `[2, 3, 5, 6, 7, 8]`

Po czwartej iteracji lista: `[2, 3, 5, 6, 7, 8]`

5. Końcowy wynik:

Po zakończeniu algorytmu lista jest w pełni posortowana: `[2, 3, 5, 6, 7, 8]`

Sortowanie z użyciem `sorted`

```python
# Funkcja obliczająca wagę słowa
def waga(slowo):
    return 2 * len(slowo)

# Przykładowe słowa
slowa = ["jabłko", "gruszka", "banan", "kiwi", "ananas"]

# Sortowanie słów na podstawie wagi
posortowane_slowa = sorted(slowa, key=waga)

# Wydrukowanie posortowanej listy
print(posortowane_slowa)
```

Sortowanie z użyciem `sorted` - 2 kryteria sortowania

```python
# Funkcja obliczająca wagę słowa
def waga(slowo):
    return 2 * len(slowo)

# Funkcja pomocnicza do sortowania
def sortuj(slowo):
    ## UWAGA (waga(slowo), slowo) - jest krotką dlatego na zewnątrz nawiasy!!!
    return (waga(slowo), slowo)

# Lista słów
lista_slow = ["jabłko", "gruszka", "banan", "kiwi", "ananas"]

# Sortowanie słów według wagi, a potem alfabetycznie
posortowane_slowo = sorted(lista_slow, key=sortuj)

# Wydrukowanie posortowanej listy
print(posortowane_slowo)
```

Liczby pierwsze

```python
def czy_pierwsza(liczba):
    if liczba == 2:
        return True
    if liczba % 2 == 0:
        return False
    od = 3
    do = liczba // 2 + 1
    # do = int(math.sqrt(liczba)) + 1
    krok = 2
    for dzielnik in range(od, do, krok):
        if liczba % dzielnik == 0:
            return False
    return True
```

```python
import math
```

```python
def sito(liczba):
    pierwsze = []
    for i in range(liczba + 1):
        if i % 2 == 1:
            pierwsze.append(True)
        else:
            pierwsze.append(False)

    pierwsze[1] = False
    pierwsze[2] = True

    od = 3
    do = int(math.sqrt(liczba)) + 1
    # do = liczba // 2 + 1
    krok = 2
    for dzielnik in range(od, do, krok):
        if pierwsze[dzielnik]:
            for i in range(dzielnik * dzielnik, liczba + 1, dzielnik):
                pierwsze[i] = False
    return pierwsze
```

NWD (największy wspólny dzielnik)

NWW (najmniejsza wspólna wielokrotność)

```python
# Funkcja do obliczenia NWD (Największy wspólny dzielnik)
def NWD(a, b):
    while b != 0:
        a, b = b, a % b
    return a  # Zwraca NWD

# Funkcja do obliczenia NWW (Najmniejsza wspólna wielokrotność)
def NWW(a, b):
    nwd = NWD(a, b)  # Obliczamy NWD
    return abs(a * b) // nwd  # Obliczamy NWW na podstawie wzoru

# Przykładowe użycie
a = 36
b = 60
nwd = NWD(a, b)
nww = NWW(a, b)

print(f'NWD({a}, {b}) = {nwd}')
print(f'NWW({a}, {b}) = {nww}')
```

Algorytmy NWD i NWW na przykładzie *a = 36* i *b = 60*

1. Algorytm NWD (Największy wspólny dzielnik)

Krok 1:
`a, b = b, a % b`

Wejście: `a = 36`, `b = 60`.
Obliczamy `a % b`:

- `b = 36 % 60 = 36` (ponieważ 36 jest mniejsze od 60).

Krok 2:
`a, b = b, a % b`

Teraz `a = 60`, `b = 36`.
Obliczamy `a % b`:

- `b = 60 % 36 = 24`.

Krok 3:
`a, b = b, a % b`

Teraz `a = 36`, `b = 24`.
Obliczamy `a % b`:

- `b = 36 % 24 = 12`.

Krok 4:
`a, b = b, a % b`

Teraz `a = 24`, `b = 12`.
Obliczamy `a % b`:

- `b = 24 % 12 = 0`.

Krok 5:

Ponieważ `b = 0`, algorytm kończy się, a ostatnie niezerowe `a` to 12.
Wynik NWD(36, 60) = 12.

2. Algorytm NWW (Najmniejsza wspólna wielokrotność)

Krok 1:
Korzystamy ze wzoru:

$$NWW(a, b) = \frac{|a \times b|}{NWD(a, b)}$$

Krok 2:
Mając NWD(36, 60) = 12, obliczamy:

$$NWW(36, 60) = \frac{|36 \times 60|}{12}$$

Krok 3:
Obliczenia:

- $36 \times 60 = 2160$
- $\frac{2160}{12} = 180$

Wynik NWW(36, 60) = 180.

Podsumowanie:

- **NWD(36, 60) = 12**
- **NWW(36, 60) = 180**

Co się zmieniło:

1. **Funkcja `NWD(a, b)`**:
   - Zawiera tylko algorytm Euklidesa, który zwraca **NWD**.
2. **Funkcja `NWW(a, b)`**:
   - Oblicza **NWW** na podstawie wcześniej obliczonego **NWD**. Korzysta z funkcji `NWD` w celu obliczenia wspólnego dzielnika, a potem stosuje wzór:

$$\text{NWW}(a, b) = \frac{|a \times b|}{\text{NWD}(a, b)}$$

Przykład działania:

Dla `a = 36` i `b = 60`:

- **NWD(36, 60)** = 12
- **NWW(36, 60)** = 180

Podsumowanie:

Teraz funkcje są rozdzielone i obie wykonują tylko jedną odpowiedzialność: obliczanie NWD oraz NWW.

Funkcje `lcm()` i `gcd()` w module `math`

LCM

- **LCM** - Least Common Multiple (NWW - Najmniejsza Wspólna Wielokrotność)
- **Opis:** Najmniejsza wspólna wielokrotność (LCM) to najmniejsza liczba, która jest wielokrotnością obu liczb.

1. `lcm(a, b)` - Najmniejsza wspólna wielokrotność (LCM)

Zwraca najmniejszą wspólną wielokrotność dwóch liczb.

```
import math
result = math.lcm(12, 15)
print(result)  # 60
```

GCD

- **GCD** - Greatest Common Divisor (NWD - Największy Wspólny Dzielnik)
- **Opis:** Największy wspólny dzielnik (GCD) to największa liczba, która dzieli obie liczby.

2. `gcd(a, b)` - Największy wspólny dzielnik (GCD)

Zwraca największy wspólny dzielnik dwóch liczb.

```
import math
result = math.gcd(12, 15)
print(result)  # 3
```