Algorytmy wyszukiwania i sortowania oraz ich zastosowanie.

Algorytmy wyszukiwania i sortowania są podstawowymi elementami w programowaniu, które znajdują szerokie zastosowanie w rozwiązywaniu problemów komputerowych. Wyszukiwanie pozwala na znalezienie określonego elementu w zbiorze danych, natomiast sortowanie umożliwia uporządkowanie danych według określonych kryteriów, co często przyspiesza kolejne operacje na tych danych.

Wprowadzenie

Algorytmy wyszukiwania

1. Liniowe (sekwencyjne)
2. Binarne

Jakie algorytmy wyszukiwania zawiera Python?

Algorytmy sortowania

1. Proste algorytmy sortowania
   - Sortowanie bąbelkowe (Bubble Sort)
   - Sortowanie przez wybieranie (Selection Sort)
2. Zaawansowane algorytmy sortowania
   - Sortowanie szybkie (Quick Sort)
   - Sortowanie przez scalanie (Merge Sort)
   - TimSort

Dlaczego algorytmy wyszukiwania i sortowania są ważne?

1. **Efektywność** - dobrze dobrany algorytm znacząco skraca czas wykonania operacji
   - bazy danych
   - systemy plików
   - aplikacje webowe
2. **Praktyczne zastosowania**:
   - przetwarzanie danych
   - analiza informacji
   - optymalizacja
   - AI
   - grafika komputerowa
3. **Podstawa dla bardziej złożonych algorytmów** - wiele z zaawansowanych algorytmów stosuje te algorytmy jako elementy składowe

Przykłady problemów, które rozwiązują algorytmy:

- Jak znaleźć najtańszy produkt w sklepie internetowym? - **wyszukiwanie minimum**
- Jak posortować listę nazwisk w porządku alfabetycznym? - **sortowanie alfabetyczne**
- Jak efektywnie odnaleźć dany rekord w dużej bazie danych? - **wyszukiwanie binarne**

---

1. Algorytmy wyszukiwania

Pod wbudowanymi funkcjami wyszukiwania w Pythonie kryją się różne algorytmy, które zostały zoptymalizowane dla wydajności w określonych strukturach danych.

Ogólny zarys wyszukiwania liniowego (sekwencyjnego)

Algorytm wyszukiwania liniowego polega na przeglądaniu wszystkich elementów listy jeden po drugim, od początku do końca, w celu znalezienia poszukiwanego elementu. Proces kończy się, gdy:

1. Znajdziemy szukany element (algorytm zwraca jego indeks).
2. Przejdziemy całą listę, nie znajdując elementu (algorytm zwraca wartość wskazującą, że element nie został znaleziony).

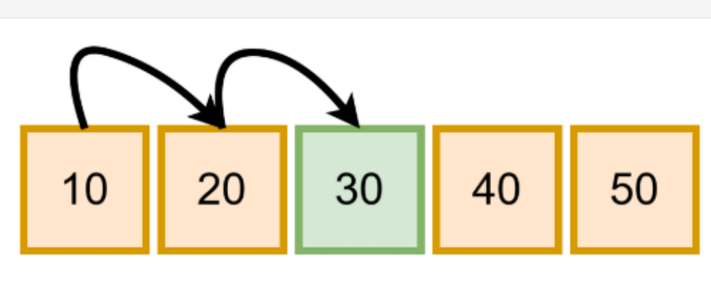
Algorytm wyszukiwania liniowego nazywa się również sekwencyjnym, ponieważ działa w sposób sekwencyjny przetwarza elementy kolekcji jeden po drugim, w ustalonej kolejności (zwykle od początku do końca).

Kroki algorytmu

1. Rozpocznij od pierwszego elementu.
2. Porównaj bieżący element z poszukiwanym.
   - Jeśli elementy są równe, zwróć indeks bieżącego elementu.
   - Jeśli nie są równe, przejdź do następnego elementu.
3. Jeśli przejdziesz całą listę i nie znajdziesz elementu, zwróć odpowiednią wartość (np. `-1` lub komunikat "nie znaleziono").

Uwaga! Algorytm wyszukiwania liniowego jest prosty i skuteczny dla niewielkich lub niesortowanych list, ale staje się nieefektywny dla dużych zbiorów danych.

In [23]:
```python
lista = [10, 20, 30, 40, 50]
szukany_element = 30
```



In [25]:
```python
def wyszukiwanie_liniowe(lista, szukany):
    for indeks, element in enumerate(lista):
        if element == szukany:
            return indeks
    return -1


indeks = wyszukiwanie_liniowe(lista, szukany_element)
if indeks != -1:
    print(f"Element {szukany_element} znaleziono na indeksie {indeks}.")
else:
    print(f"Element {szukany_element} nie został znaleziony.")
```

```
Element 30 znaleziono na indeksie 2.
```

Ogólny zarys wyszukiwania binarnego

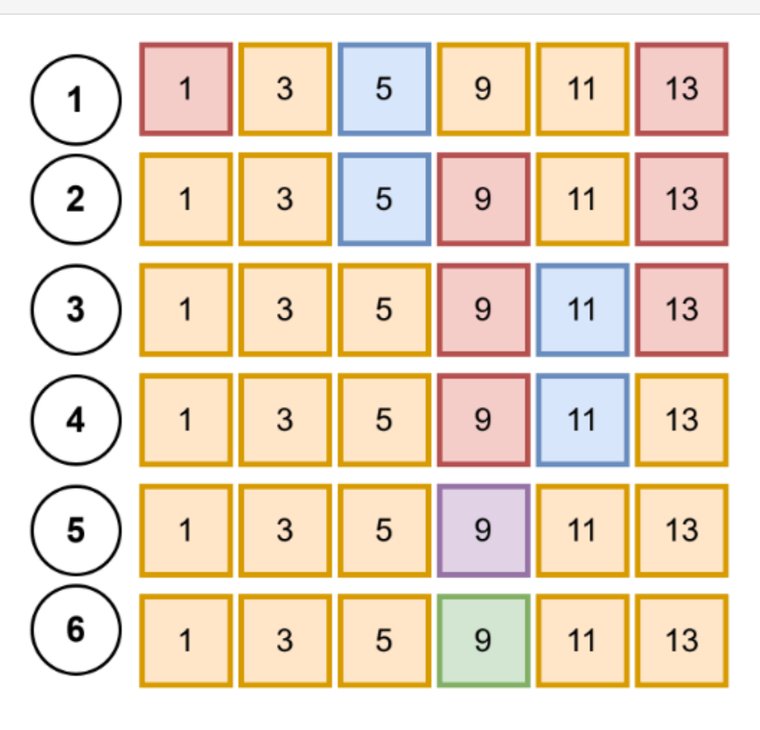
Algorytm stosowany do znajdowania elementu w posortowanej liście. Zamiast przeglądać każdy element z kolei (jak w wyszukiwaniu liniowym), algorytm dzieli listę na pół przy każdym kroku, eliminując połowę możliwych wyników. Dzięki temu złożoność czasowa jest znacznie niższa niż w przypadku wyszukiwania liniowego.

Kroki algorytmu

1. **Warunek wstępny**: Lista musi być posortowana.
2. Ustal indeksy:
   - `lewy` - początek listy (indeks 0).
   - `prawy` - koniec listy (indeks n-1).
3. Znajdź środkowy element:
   - Oblicz indeks środka: `srodek = (lewy + prawy) // 2`.
4. Porównaj środkowy element z poszukiwanym:
   - Jeśli jest równy, zwróć jego indeks.
   - Jeśli poszukiwany element jest mniejszy, przeszukuj lewą połowę (ustaw `prawy = srodek - 1`).
   - Jeśli jest większy, przeszukuj prawą połowę (ustaw `lewy = srodek + 1`).
5. Powtarzaj kroki 3-4, aż znajdziesz element lub przeszukiwany zakres stanie się pusty (`lewy > prawy`).

Uwaga, jeśli lista nie jest posortowana, algorytm zwróci błędne wyniki lub nie znajdzie elementu. Dlatego sortowanie jest koniecznym krokiem przed zastosowaniem wyszukiwania binarnego.

In [40]:
```python
lista = [1, 3, 5, 9, 11, 13]
szukany_element = 9
```



In [42]:
```python
def wyszukiwanie_binarne(lista, szukany):
    lewy, prawy = 0, len(lista) - 1

    while lewy <= prawy:
        srodek = (lewy + prawy) // 2
        if lista[srodek] == szukany:
            return srodek
        elif lista[srodek] < szukany:
            lewy = srodek + 1
        else:
            prawy = srodek - 1

    return -1


indeks = wyszukiwanie_binarne(lista, szukany_element)
if indeks != -1:
    print(f"Element {szukany_element} znaleziono na indeksie {indeks}.")
else:
    print(f"Element {szukany_element} nie został znaleziony.")
```

```
Element 9 znaleziono na indeksie 3.
```

1.1. Wyszukiwanie w listach

Wyszukiwanie w listach w Pythonie zazwyczaj opiera się na **wyszukiwaniu liniowym**:

- `in` i `index()` - implementowane jako iteracyjne przeglądanie elementów listy od początku do końca.
- Algorytm:
  1. Rozpocznij od pierwszego elementu.
  2. Porównuj każdy element z poszukiwanym.
  3. Zatrzymaj się, gdy znajdziesz element lub dojdziesz do końca listy.
- Złożoność czasowa tego algorytmu to $O(n)$, gdzie n to liczba elementów w liście.

1.2. Wyszukiwanie w słownikach

Słowniki w Pythonie wykorzystują **tablice mieszające (hash tables)**:

- **Algorytm**:
  1. Klucz jest przekształcany za pomocą funkcji mieszającej (`hash()`).
  2. Wynik funkcji określa miejsce w tablicy, gdzie dane są przechowywane.
  3. Odczyt jest bezpośredni (czas stały $O(1)$), o ile nie wystąpi konflikt.
  4. W przypadku konfliktów (dwa różne klucze mają ten sam wynik funkcji mieszającej), Python stosuje rozwiązania takie jak listy łańcuchowe (chaining).
- **Złożoność czasowa**:
  - Średnio $O(1)$ dla wyszukiwania klucza.
  - W najgorszym przypadku $O(n)$, jeśli wystąpią liczne konflikty.

Wyszukiwanie w wartościach (np. `x in slownik.values()`) wymaga przeglądania całej kolekcji i ma złożoność $O(n)$.

1.3. Wyszukiwanie w zbiorach

Zbiory (`set`) w Pythonie również wykorzystują **tablice mieszające**, działając na podobnych zasadach jak słowniki:

- Operacje takie jak `x in zbior` są zoptymalizowane do średniego czasu $O(1)$.
- Konflikty mieszania są obsługiwane podobnie jak w słownikach.

1.4. Wyszukiwanie binarne (`bisect`)

Moduł `bisect` opiera się na **algorytmie wyszukiwania binarnego**, który wymaga posortowanej listy:

- **Algorytm**:
  1. Porównaj poszukiwany element ze środkowym elementem listy.
  2. Jeśli element jest mniejszy, szukaj w lewej połowie; jeśli większy, szukaj w prawej połowie.
  3. Powtarzaj, aż znajdziesz element lub lista zostanie podzielona do pustego zbioru.
- Złożoność czasowa to $O(log n)$.

Funkcje `bisect_left` i `bisect_right` precyzują, czy należy zwrócić pierwsze czy ostatnie wystąpienie elementu.

1.5. Wyszukiwanie w ciągach znaków

a) `in` oraz `find()`

Te operacje korzystają z optymalizowanych wersji wyszukiwania podciągu:

- Python używa algorytmu **Knutha-Morrisa-Pratta (KMP)** i podobnych metod opartych na automatach skończonych.
- **Algorytm KMP**:
  1. Tworzy się tablicę "przesunięć" na podstawie wzorca (podciągu).
  2. Przy porównywaniu ciągu i wzorca przeskakuje się pewne elementy, jeśli wystąpi niezgodność, dzięki tablicy przesunięć.
  3. To znacznie redukuje liczbę porównań w porównaniu z naiwnym podejściem.
- **Złożoność czasowa**:
  - Tworzenie tablicy przesunięć: $O(m)$, gdzie m to długość wzorca.
  - Wyszukiwanie $O(n + m)$, gdzie n to długość ciągu, a m długość wzorca.

Algorytm KMP

Knuth-Morris-Pratt (KMP) to algorytm wyszukiwania wzorca w tekście, który eliminuje konieczność powtarzania porównań już sprawdzonych znaków. Dzięki temu działa szybciej niż klasyczne (naiwne) podejście.

1. Budowa tablicy LPS (Longest Prefix Suffix)

Tablica LPS zawiera dla każdego znaku wzorca długość najdłuższego prefiksu, który jest równocześnie sufiksem dla fragmentu wzorca kończącego się na tym znaku.

- **prefiks** - to początkowa część ciągu znaków
- **sufiks** - to końcowa część ciągu znaków

Jeśli mamy określony wzorzec przykładowo ciag znaków `ABABC`, prefiksy i sufiksy dla tego wzorca wyglądają następująco:

- fragment `ABA` - prefiksy `['A', 'AB']`, sufiksy `['A', 'BA']`
- najdłuższy sufkis, który jest również prefiksem `A`

Tablica LPS jest stosowana aby:

- wyeliminować powtarzające się porównania
- zastosować wiedzę o strukturze wzorca
- zoptymalizować czas wyszukiwania

Budowanie tablicy LPS

1. Pierwszym krokiem jest stworzenie tablicy o długości wzorca, wypełnionej zerami. Dla podanego wcześniej przykładu wygląda to tak: `[0, 0, 0, 0, 0]`
2. Dla każdego znaku wzorca porównujemy znaki na początku i końcu wzorca:
   - Jeżeli się zgadzają, zwiększamy długość prefiksu i zapisujemy ją w tablicy LPS
   - Jeżeli nie, cofamy się do wcześniejszego dopasowania tablicy LPS

Wartości tablicy LPS, reprezentują długości najdłuższego prefiksu wzorca, który jest jednocześnie jego sufiksem, dla każdego znaku wzorca. Każda wartość w tablicy wskazuje, jak daleko można przesunąć wzorzec po niezgodności podczas wyszukiwania, bez konieczności porównywania znaków od początku.

In [50]:
```python
wzorzec = "ABABAC"
```

Indeks (i)	Fragment wzorca	Prefiksy	Sufiksy	LPS[i]
0	A	[]	[]	0
1	AB	["A"]	["B"]	0
2	ABA	["A", "AB"]	["A", "BA"]	1
3	ABAB	["A", "AB", "ABA"]	["B", "AB", "BAB"]	2
4	ABABA	["A", "AB", "ABA"]	["A", "BA", "ABA"]	3
5	ABABAC	["A", "AB", "ABA"]	["C", "AC", "BAC"]	0

1. **Indeks 0 (A).** Nie ma prefiksu, który byłby sufiksem – wartość (LPS[0] = 0).
2. **Indeks 1 (AB).** Nie ma wspólnego prefiksu i sufiksu – wartość (LPS[1] = 0).
3. **Indeks 2 (ABA).** Najdłuższy prefiks to `A`, który jest jednocześnie sufiksem – wartość (LPS[2] = 1).
4. **Indeks 3 (ABAB).** Najdłuższy prefiks to `AB`, który jest jednocześnie sufiksem – wartość (LPS[3] = 2).
5. **Indeks 4 (ABABA).** Najdłuższy prefiks to `ABA`, który jest jednocześnie sufiksem – wartość (LPS[4] = 3).
6. **Indeks 5 (ABABAC).** Brak wspólnego prefiksu i sufiksu – wartość (LPS[5] = 0).

In [52]:
```python
def oblicz_lps(wzorzec):
    m = len(wzorzec)
    lps = [0] * m
    dlugosc = 0
    i = 1

    while i < m:
        if wzorzec[i] == wzorzec[dlugosc]:
            dlugosc += 1
            lps[i] = dlugosc
            i += 1
        else:
            if dlugosc != 0:
                dlugosc = lps[dlugosc - 1]
            else:
                lps[i] = 0
                i += 1
    return lps
```

In [54]:
```python
oblicz_lps(wzorzec)
```

Out[54]:
```
[0, 0, 1, 2, 3, 0]
```

2. Wyszukiwanie wzorca w tekście

Po zbudowaniu tablicy LPS przeszukujemy tekst:

- Porównujemy wzorzec z tekstem znak po znaku.
- Jeśli napotykamy niezgodność:
  - Zamiast przesuwać wzorzec na sam początek, używamy tablicy LPS, aby przeskoczyć część wzorca, która już pasowała.

In [84]:
```python
tekst = "ABABABABAC"
```

Dane wejściowe:

- **Tekst (T):** `ABABABABAC`
- **Wzorzec (P):** `ABABAC`
- **Tablica LPS:** `[0, 0, 1, 2, 3, 0]`

Inicjalizacja:

1. **Indeksy:**
   - `i = 0` – indeks w tekście.
   - `j = 0` – indeks w wzorcu.
2. Rozpoczynamy porównywanie od początku wzorca i tekstu.

Krok 1. Porównanie pierwszego znaku wzorca z tekstem

- ($T[i] = T[0] = A$), ($P[j] = P[0] = A$).
- Znaki pasują.
- Zwiększamy oba indeksy:
  - ($i = 1$), ($j = 1$).

Krok 2. Porównanie drugiego znaku wzorca z tekstem

- ($T[i] = T[1] = B$), ($P[j] = P[1] = B$).
- Znaki pasują.
- Zwiększamy oba indeksy:
  - ($i = 2$), ($j = 2$).

Krok 3. Porównanie trzeciego znaku wzorca z tekstem

- ($T[i] = T[2] = A$), ($P[j] = P[2] = A$).
- Znaki pasują.
- Zwiększamy oba indeksy:
  - ($i = 3$), ($j = 3$).

Krok 4. Porównanie czwartego znaku wzorca z tekstem

- ($T[i] = T[3] = B$), ($P[j] = P[3] = B$).
- Znaki pasują.
- Zwiększamy oba indeksy:
  - ($i = 4$), ($j = 4$).

Krok 5. Porównanie piątego znaku wzorca z tekstem

- ($T[i] = T[4] = A$), ($P[j] = P[4] = A$).
- Znaki pasują.
- Zwiększamy oba indeksy:
  - ($i = 5$), ($j = 5$).

Krok 6. Porównanie szóstego znaku wzorca z tekstem

- ($T[i] = T[5] = B$), ($P[j] = P[5] = C$).
- **Niezgodność!**
  - Zamiast zaczynać od początku wzorca, korzystamy z tablicy LPS:
    - ($j = LPS[j-1] = LPS[4] = 3$).
  - To oznacza, że końcowe `ABA` we wzorcu jest również jego prefiksem. Przesuwamy wzorzec tak, aby `ABA` pasowało do tekstu.

Krok 7. Kontynuacja po przesunięciu wzorca

- (i) pozostaje bez zmian: ($i = 5$).
- Porównujemy wzorzec od pozycji ($j = 3$).

1. Porównanie:
   - ($T[i] = T[5] = B$), ($P[j] = P[3] = B$).
   - Znaki pasują.
   - ($i = 6$), ($j = 4$).
2. Porównanie:
   - ($T[i] = T[6] = A$), ($P[j] = P[4] = A$).
   - Znaki pasują.
   - ($i = 7$), ($j = 5$).
3. Porównanie:
   - ($T[i] = T[7] = B$), ($P[j] = P[5] = C$).
   - **Niezgodność!**
   - Korzystamy z tablicy LPS:
     - ($j = LPS[4] = 3$).

Krok 8. Kolejne przesunięcie wzorca

1. (i) pozostaje bez zmian: ($i = 7$).
2. Porównujemy od ($j = 3$).
3. Porównanie:
   - ($T[i] = T[7] = B$), ($P[j] = P[3] = B$).
   - Znaki pasują.
   - ($i = 8$), ($j = 4$).
4. Porównanie:
   - ($T[i] = T[8] = A$), (P[j] = P[4] = A).
   - Znaki pasują.
   - ($i = 9$), ($j = 5$).
5. Porównanie:
   - ($T[i] = T[9] = C$), ($P[j] = P[5] = C$).
   - Znaki pasują.
   - ($i = 10$), ($j = 6$).

Krok 9. Dopasowanie zakończone

- ($j = 6$), co oznacza, że cały wzorzec został dopasowany.
- Pozycja początkowa dopasowania w tekście:
  - ($i - j = 10 - 6 = 4$).

In [87]:
```python
def kmp(tekst, wzorzec):

    n = len(tekst)
    m = len(wzorzec)
    lps = oblicz_lps(wzorzec)

    i = 0
    j = 0

    while i < n:
        if tekst[i] == wzorzec[j]:
            i += 1
            j += 1

        if j == m:
            print(f"Wzorzec znaleziono na indeksie {i - j}")
            j = lps[j - 1]
        elif i < n and tekst[i] != wzorzec[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
```

In [89]:
```python
kmp(tekst, wzorzec)
```

```
Wzorzec znaleziono na indeksie 4
```

b) `startswith()` i `endswith()`

Metody te są zoptymalizowane do szybkich operacji na początku i końcu ciągów. Pod spodem wykonują sprawdzenie porównawcze bez iterowania przez cały ciąg.

Zestawienie algorytmów w Python

Struktura danych	Operacja	Algorytm	Złożoność czasowa
Lista	`in`, `index()`	Wyszukiwanie liniowe	$O(n)$
Słownik	`in`, `get()`	Tablica mieszająca	$O(1)$
Zbiór	`in`	Tablica mieszająca	$O(1)$
Posortowana lista	`bisect`	Wyszukiwanie binarne	$O(log n)$
Ciąg znaków	`in`, `find()`, `index()`	Knuth-Morris-Pratt (KMP)	$O(n + m)$

Źródła informacji

Informacje o algorytmach wbudowanych w Pythonie pochodzą z następujących źródeł i dokumentacji:

Dokumentacja języka Python:

- Wyszukiwanie w strukturach danych Python Data Structures
- Moduł `bisect` Bisect module documentation
- Operacje na ciągach znaków String methods
- Algorytm KMP Knuth–Morris–Pratt

2. Algorytmy sortowania

Algorytmy sortowania to podstawowe narzędzia umożliwiające uporządkowanie danych w określonym porządku (np. rosnącym lub malejącym). W Pythonie wiele z tych algorytmów zostało zaimplementowanych i zoptymalizowanych w bibliotece standardowej, ale warto znać podstawowe techniki, które kryją się za tymi funkcjami.

1.1. Sortowanie bąbelkowe

Sortowanie bąbelkowe polega na wielokrotnym porównywaniu sąsiednich elementów w liście i zamianie ich miejscami, jeśli są w niewłaściwej kolejności. Proces ten jest powtarzany, aż do momentu, gdy lista zostanie całkowicie posortowana.

Kroki algorytmu

1. Rozpocznij od pierwszego elementu.
2. Porównaj bieżący element z następnym.
   - Jeśli są w złej kolejności, zamień je miejscami.
3. Przejdź do następnej pary elementów i powtórz krok 2.
4. Po zakończeniu jednego przebiegu sprawdź, czy wystąpiły zamiany:
   - Jeśli nie, lista jest posortowana.
   - Jeśli tak, rozpocznij kolejny przebieg.
5. Powtarzaj kroki 1–4, aż lista będzie posortowana.

Uwaga! Sortowanie bąbelkowe jest bardzo nieefektywne dla dużych zbiorów danych ze względu na swoją złożoność czasową $O(n^2)$.

1.2. Sortowanie przez wstawianie

Sortowanie przez wstawianie działa w sposób podobny do układania kart w ręku. Każdy element jest pobierany z nieposortowanej części i wstawiany w odpowiednie miejsce w części posortowanej.

Kroki algorytmu

1. Rozpocznij od drugiego elementu (pierwszy element jest traktowany jako posortowany).
2. Pobierz bieżący element i znajdź jego miejsce w posortowanej części listy.
3. Przesuń elementy większe od bieżącego w prawo, aby zrobić miejsce na wstawienie.
4. Wstaw bieżący element w odpowiednie miejsce.
5. Powtarzaj kroki 2–4 dla każdego kolejnego elementu.

Uwaga! Złożoność czasowa wynosi $O(n^2)$ w przypadku najgorszego scenariusza.

1.3. Sortowanie szybkie (quicksort)

Quicksort to jeden z najszybszych algorytmów sortowania ogólnego przeznaczenia. Działa na zasadzie "dziel i zwyciężaj", dzieląc listę na mniejsze podlisty i sortując je niezależnie.

Kroki algorytmu

1. Wybierz element z listy jako tzw. pivot (np. pierwszy, ostatni, lub losowy element).
2. Podziel listę na trzy części:
   - Elementy mniejsze od pivotu.
   - Pivot.
   - Elementy większe od pivotu.
3. Rekurencyjnie zastosuj quicksort na podlistach mniejszych i większych.
4. Połącz posortowane podlisty i pivot w jedną całość.

Uwaga! Quicksort jest bardzo efektywny ze średnią złożonością czasową $O(n \log n)$, ale w najgorszym przypadku może działać w czasie $O(n^2)$ (np. gdy pivot jest źle dobrany).

1.4. Algorytm TimSort (`sorted()` oraz `.sort()` na listach)

TimSort to hybrydowy algorytm sortowania używany jako domyślny wbudowany mechanizm sortowania w Pythonie (oraz w Javie). Został zaprojektowany jako połączenie **sortowania przez wstawianie** i **sortowania przez scalanie**, aby optymalizować wydajność na rzeczywistych danych, które często zawierają częściowo posortowane sekwencje.

1.4.1. Główne cechy TimSort

- Algorytm dzieli dane na "runy" – podlisty, które są już posortowane lub można je szybko posortować.
- Dla małych runów używane jest sortowanie przez wstawianie.
- Posortowane runy są łączone w większe runy przy użyciu sortowania przez scalanie.

1.4.2. Kroki algorytmu TimSort

1. **Podział danych na runy:**
   - Algorytm przeszukuje dane, aby zidentyfikować naturalnie posortowane fragmenty (rosnące lub malejące).
   - Dla fragmentów malejących kolejność jest odwracana, aby uzyskać runy w porządku rosnącym.
   - Jeśli run jest zbyt krótki, zostaje rozszerzony za pomocą sortowania przez wstawianie, aby osiągnąć minimalny rozmiar (domyślnie 32 w Pythonie).

In [ ]:
```python
def znajdz_run(tablica, poczatek, n):
    koniec = poczatek + 1
    if koniec == n:
        return koniec

    if tablica[koniec] < tablica[poczatek]:
        while koniec < n and tablica[koniec] < tablica[koniec - 1]:
            koniec += 1
        tablica[poczatek:koniec] = reversed(tablica[poczatek:koniec])
    else:
        while koniec < n and tablica[koniec] >= tablica[koniec - 1]:
            koniec += 1

    return koniec
```

2. **Sortowanie runów:**
   - Mniejsze runy są sortowane za pomocą sortowania przez wstawianie.

In [ ]:
```python
def sortowanie_przez_wstawianie(tablica, lewy, prawy):
    for i in range(lewy + 1, prawy + 1):
        klucz = tablica[i]
        j = i - 1
        while j >= lewy and tablica[j] > klucz:
            tablica[j + 1] = tablica[j]
            j -= 1
        tablica[j + 1] = klucz
```

3. **Scalanie runów:**
   - Posortowane runy są łączone parami w większe runy za pomocą sortowania przez scalanie.
   - Algorytm utrzymuje równowagę między rozmiarami runów, aby uniknąć nieefektywności (np. jeśli runy są zbyt małe lub zbyt nierówne).

In [ ]:
```python
def scal(tablica, lewy, srodek, prawy):
    lewa_podlista = tablica[lewy:srodek + 1]
    prawa_podlista = tablica[srodek + 1:prawy + 1]

    i, j, k = 0, 0, lewy

    while i < len(lewa_podlista) and j < len(prawa_podlista):
        if lewa_podlista[i] <= prawa_podlista[j]:
            tablica[k] = lewa_podlista[i]
            i += 1
        else:
            tablica[k] = prawa_podlista[j]
            j += 1
        k += 1

    while i < len(lewa_podlista):
        tablica[k] = lewa_podlista[i]
        i += 1
        k += 1

    while j < len(prawa_podlista):
        tablica[k] = prawa_podlista[j]
        j += 1
        k += 1
```

4. **Główna funkcja TimSort, uwzględniająca wszystkie kroki algorytmu.**

In [ ]:
```python
def tim_sort(tablica):
    n = len(tablica)
    RUN = 32

    runy = []
    poczatek = 0

    while poczatek < n:
        koniec = znajdz_run(tablica, poczatek, n)
        if koniec - poczatek < RUN:
            sortowanie_przez_wstawianie(tablica, poczatek, min(poczatek + RUN - 1, n - 1))
            koniec = min(poczatek + RUN, n)
        runy.append((poczatek, koniec - 1))
        poczatek = koniec

    while len(runy) > 1:
        nowe_runy = []
        for i in range(0, len(runy) - 1, 2):
            lewy, srodek = runy[i]
            _, prawy = runy[i + 1]
            scal(tablica, lewy, srodek, prawy)
            nowe_runy.append((lewy, prawy))
        if len(runy) % 2 == 1:
            nowe_runy.append(runy[-1])
        runy = nowe_runy
```

1.4.3. Zalety TimSort

- Algorytm został zaprojektowany z myślą o danych częściowo posortowanych, które często występują w praktyce.
- TimSort jest stabilny, co oznacza, że elementy o tej samej wartości zachowują pierwotną kolejność.
- Złożoność czasowa:
  - Średnia i najlepsza: $O(n \log n)$.
  - Najgorsza: $O(n \log n)$, dzięki starannej konstrukcji algorytmu.
- Scalanie odbywa się w sposób zoptymalizowany pod kątem pamięci.

Źródło informacji

- Wyjaśnienie TimSort, jego działania oraz implementacji. GeeksforGeeks
- Historia, działanie i zastosowania algorytmu TimSort. Wikipedia
- Zbiór algorytmów sortujących, ich opis i złożoność. eduinf.waw.pl
- Wykład o zaawansowanych algorytmach sortowania, w tym TimSort. Uniwersytet Wrocławski
- Proste wyjaśnienie podstawowych algorytmów sortowania. Daniel Jeziorski
- Analiza i szczegóły dotyczące TimSort, połączenia sortowania przez scalanie i wstawianie. Kirupa
- Film opisujący szczegóły działania TimSort. YouTube
- Przykłady i wyjaśnienia algorytmów sortowania. Uniwersytet Morski

Zadania do realizacji

Uwaga! W zadaniach 1 - 6 proszę nie korzystać z wbudowanych mechanizmów sortowania i wyszukiwania.

Funkcja mierząca czas wykonywania

```python
import time

def zmierz_czas_sortowania(func, lista):

    start = time.time()
    func()
    koniec = time.time()
    return koniec - start
```

1. Algorytmy wyszukiwania

1. Zmodyfikuj funkcję wyszukiwania liniowego tak, aby zwracała wszystkie indeksy wystąpień szukanego elementu.
2. Przetestuj działanie wyszukiwania binarnego na posortowanych i niesortowanych listach. Wyjaśnij różnice w wynikach w formie komentarza.
3. Napisz program, który pozwala wyszukać nazwisko w tej liście za pomocą wyszukiwania binarnego. Dla porównania, zaimplementuj wyszukiwanie liniowe i zmierz czas działania obu algorytmów. Nazwiska należy zaimportować z pliku `nazwiska_posortowane.txt` i zapisać je do listy.

---

2. Algorytmy sortowania

4. Napisz funkcję implementującą sortowanie bąbelkowe. Porównaj jej działanie na małej (np. 10 elementów) i dużej (np. 1000 elementów) liście. Algorytm ma zakończyć działanie działanie, gdy podczas przebiegu nie nastąpi żadna zamiana elementów.
5. Przetestuj algorytm sortowania przez wstawianie na liście częściowo posortowanej i całkowicie losowej. Porównaj czas działania w obu przypadkach.

```python
def generuj_liste(rozmiar, zakres):
        return [random.randint(0, zakres) for _ in range(rozmiar)]
```

6. Napisz funkcję implementującą algorytm sortowania szybkiego. Użyj pierwszego elementu jako pivotu.

Zadania dodatkowe

7. Napisz program porównujący czas działania różnych algorytmów sortowania (np. bąbelkowego, przez wstawianie, szybkiego) dla list o różnych rozmiarach (np. 10, 100, 1000 elementów).
8. Otrzymujesz listę liczb. Użyj algorytmu sortowania, aby znaleźć element najczęściej występujący w liście.
9. Napisz program, który generuje losową listę, sortuje ją i pozwala użytkownikowi wyszukiwać elementy za pomocą wyszukiwania binarnego.
10. Przygotuj listy o różnych strukturach (np. losowe, częściowo posortowane, odwrotnie posortowane) i zbadaj, jak różne algorytmy sortowania radzą sobie z nimi pod kątem czasu działania.
11. Przygotuj grę, w której gracze muszą ręcznie uporządkować listę liczb według określonego algorytmu (np. bąbelkowego). Sprawdź, czy stosują właściwe kroki algorytmu.

In [ ]: