Sortowanie kubełkowe w języku Python

Źródło: Amy Parkes, domena publiczna.

Sortowanie kubełkowe jest jednym z najbardziej intuicyjnych sortowań używanych w życiu codziennym. Polega ono na umieszczaniu wartości w odpowiednich „szufladkach” (tak zwanych kubełkach), przy założeniu, że każdy kubełek zawiera elementy o tej samej wartości. W tym e-materiale dowiesz się, jak zaimplementować algorytm sortowania kubełkowego w języku programowania Python.

Więcej informacji o sortowaniu kubełkowym znajdziesz w e-materiale Sortowanie kubełkowe.

Implementacje sortowania kubełkowego w innych językach przedstawiamy w e-materiałach:

- Sortowanie kubełkowe w języku C++,
- Sortowanie kubełkowe w języku Java.

Więcej zadań? Sprawdź e-materiał: Sortowanie kubełkowe – zadania maturalne.

> **Twoje cele**
>
> - Przeanalizujesz implementację algorytmu sortowania kubełkowego liczb rzeczywistych oraz całkowitych w języku Python.
> - Zaimplementujesz algorytm sortowania kubełkowego w języku Python.

- Rozwiążesz ćwiczenia sprawdzające twoją wiedzę dotyczącą algorytmu sortowania kubełkowego.

Przeczytaj

> **Polecenie 1**
>
> Zapoznaj się z animacją przedstawiającą sortowanie kubełkowe.

Film dostępny pod adresem /preview/resource/Rcz1Xt1KeUHUU

Film mówiący o sortowaniu kubełkowym.

> **Polecenie 2**
>
> Prześledź działanie algorytmu w praktyce. W tym celu przygotuj zestaw przedmiotów do posortowania. Mogą to być żetony, korki, kapsle czy inne małe przedmioty, mające charakterystyczną cechę, według której można je posortować.
>
> Następnie ułóż je w losowej kolejności w jednej linii. Powstanie w ten sposób zbiór, który posortujesz.
>
> Wszystkie ułożone w linii przedmioty pogrupuj w zbiory. Elementy można zidentyfikować poprzez ich charakterystyczną cechę. W przykładzie przedstawionym w animacji był to rodzaj przyboru szkolnego. W twoim przypadku może być to kolor, rozmiar czy inne dowolne kryterium.
>
> Następnie wybierz zbiór i zacznij układać jego elementy, jeden po drugim, w linii. Gdy dany zbiór się skończy, wybierz następny i powtarzaj proces. Przedłużaj układaną linię aż do momentu, gdy skończą się wszystkie zbiory.

Sortowanie kubełkowe

Algorytm sortowania kubełkowego polega na zliczeniu wystąpień danych wartości w liście wejściowej, a następnie na wygenerowaniu listy wyjściowej, zawierającej dane w odpowiednich kubełkach, posortowane zgodnie z ich liczbą.

Sortowanie liczb całkowitych

Specyfikacja problemu:

Dane:

- `liczby` – nieposortowana lista liczb całkowitych zawierająca dane do posortowania

Wynik:

- `posortowane_elem` – posortowana niemalejąco lista liczb całkowitych

Zaczynamy od odnalezienia najmniejszej i największej liczby należącej do listy wejściowej.

> **Ważne!**
>
> Aby odnaleźć odpowiednie wartości, możemy skorzystać z wbudowanych funkcji `min()` oraz `max()`.

```
# znajdowanie wartości minimalnej i maksymalnej
min_elem = min(liczby)
max_elem = max(liczby)
```

W tym e-materiale wykorzystamy jednak implementację zaprezentowaną w e-materiale Sortowanie kubełkowe.

```
n = len(liczby)
min_elem = liczby[0]
max_elem = liczby[0]
for i in range(1, n):
    if liczby[i] < min_elem:
        min_elem = liczby[i]
    if liczby[i] > max_elem:
        max_elem = liczby[i]
```

Tworzy listę zliczającą `kubelek` o długości `max_elem - min_elem + 1`, tak by każda liczba z przedziału `<min_elem, max_elem>` miała swój indeks, a więc kubełek. Lista służy do zliczania, ile razy każda liczba wystąpi w przedziale `<min_elem, max_elem>`.

Zerujemy tę listę, żeby mieć pewność co do poprawności wykonania kodu.

```
# przygotowanie listy zliczającej
k = max_elem - min_elem + 1
kubelek = [0] * k
```

Iterujemy po wszystkich elementach z listy `liczby` i zwiększamy o jeden wartość dla odpowiedniego indeksu w liście zliczającej `kubelek`. Indeks jest obliczany jako różnica między wartością liczby a minimalną liczbą.

```
# zliczanie wartości
for elem in liczby:
    kubelek[elem - min_elem] += 1
```

Następnie iterujemy po wszystkich indeksach w liście zliczającej i tworzymy nową listę: `posortowane_elem`, dodając liczbę odpowiadającą temu indeksowi tyle razy, ile jest zliczona w liście zliczającej.

Na koniec program zwraca posortowaną listę `posortowane_elem`.

```
# porządkowanie
posortowane_elem = []
for i in range(k):
    while kubelek[i] > 0:
        posortowane_elem.append(i + min_elem)
        kubelek[i] -= 1
return posortowane_elem
```

Kompletny kod programu:

```
def sort_kubelk(liczby):
    # znajdowanie wartości minimalnej i maksymalnej
    n = len(liczby)
    min_elem = liczby[0]
    max_elem = liczby[0]
    for i in range(1, n):
        if liczby[i] < min_elem:
            min_elem = liczby[i]
        if liczby[i] > max_elem:
            max_elem = liczby[i]
    # przygotowanie listy zliczającej
    k = max_elem - min_elem + 1
    kubelek = [0] * k
    # zliczanie wartości
    for elem in liczby:
        kubelek[elem - min_elem] += 1
    # porządkowanie
    posortowane_elem = []
    for i in range(k):
        while kubelek[i] > 0:
            posortowane_elem.append(i + min_elem)
            kubelek[i] -= 1
    return posortowane_elem

liczby = [3, 2, 3, 4, -4, 2, 3, 2, 0, -5]
print(sort_kubelk(liczby))

# posortowana lista
# [-5, -4, 0, 2, 2, 2, 3, 3, 3, 4]
```

Widzimy, że w wyniku działania algorytmu poszczególne wystąpienia liczb są ułożone w kolejności od najmniejszej do największej.

> **Ważne!**
>
> Aby uporządkować liczby nierosnąco, należy listę liczb odczytywać od końca. Zmieni się więc tylko kod drugiej pętli:

```
1 for i in range(k - 1, -1, -1):
```

> Sortowanie kubełkowe jest sortowaniem stabilnym (utrzymuje kolejność występowania elementów o tym samym kluczu); wymaga zadeklarowania dodatkowej pamięci do sortowania.

Sortowanie liczb rzeczywistych

Liczby rzeczywiste również mogą być porządkowane z użyciem algorytmu sortowania kubełkowego. Będzie się on jednak różnił od wersji dla liczb całkowitych.

W przypadku algorytmu dla liczb całkowitych każda liczba ma własny osobny kubełek, który zawiera licznik jej wystąpień. Natomiast podczas sortowania liczb rzeczywistych kubełki są przedziałami, do których trafiają liczby należące do danego zakresu. Przedziały wyznaczamy, dzieląc maksymalny element listy przez liczbę elementów.

W implementacji wykorzystamy taką samą liczbę kubełków (przedziałów), jak w wersji algorytmu dla liczb całkowitych. Wartości, które trafią do danego kubełka, będą się mieścić w przedziałach `[x, x + 1)` dla liczb nieujemnych i `(x - 1, x]` dla liczb ujemnych. Liczba `3,7` trafi do kubełka reprezentującego przedział `[3; 4)`, a liczba `-4,2` do przedziału `(-5; -4]`. Możemy wywołać funkcję również z inną liczbą kubełków – wartości w liście zostaną odpowiednio uporządkowane dzięki wyznaczeniu przedziałów w implementacji.

Przeanalizujmy program sortujący niemalejąco liczby zmiennoprzecinkowe metodą kubełkową.

Specyfikacja problemu:

Dane:

- `n` – liczba naturalna; liczba kubełków
- `liczby` – nieposortowana lista nieujemnych liczb rzeczywistych zawierająca dane do posortowania

Wynik:

- posortowana niemalejąco lista liczb

```
def sort_kubelk(liczby, n):
    # Tworzymy listę kubełków o rozmiarze n
    kubelki = [[] for _ in range(n)]
    # Przypisujemy liczbę do odpowiedniego kubełka
    for elem in liczby:
        kubelek_indeks = int(elem * n)
        kubelki[kubelek_indeks].append(elem)
    # Sortujemy każdy kubełek za pomocą sortowania bąbelkowego
    for i in range(0, n - 1):
        for j in range(1, n - i):
            if liczby[j - 1] < liczby[j]:
                liczby[j], liczby[j - 1] = liczby[j - 1], liczby[
    # Inicjalizujemy zmienną do przechowywania wyniku
    posortowane_liczby = []
    # Przeglądamy kubełki i dodajemy liczby do wyniku
    for kubelek in kubelki:
        for elem in kubelek:
            posortowane_liczby.append(elem)
    return posortowane_liczby

liczby = [0.1, 0.4, 0.2, 0.9, 0.5, 0.8, 0.3, 0.7, 0.6]
n = len(liczby)
print(sort_kubelk(liczby, n))
```

> **Ważne!**
>
> Stworzymy dodatkową funkcję `sort_babelkowo()`, którą zrealizujemy sortowanie elementów wewnątrz kubełka. Wykorzystamy ją w trakcie działania funkcji `sort_kubelk()`.

```
def sort_babelkowo(liczby):
    n = len(liczby)
    for i in range(0, n - 1):
        for j in range(1, n - i):
            if liczby[j - 1] > liczby[j]:
                liczby[j], liczby[j - 1] = liczby[j - 1], liczb
    return liczby
```

Implementujemy funkcję `sort_kubelk` przyjmującą jako parametry listę liczb zmiennoprzecinkowych `liczby` oraz liczbę kubełków `n`.

```
def sort_kubelk(liczby, n):
```

Tworzymy listę `kubelki` o rozmiarze `n+1`, aby zapobiec błędom `IndexError: list assignment index out of range`. Służy ona do przechowywania liczb w odpowiednich kubełkach. Używamy standardowego generatora listowego w Pythonie.

```
# Tworzymy listę kubełków o rozmiarze n+1
kubelki = [[] for _ in range(n+1)]
```

Dla każdej liczby z wejściowej listy obliczamy indeks kubełka, dzieląc całkowicie dany element listy (`elem`) przez rozmiar kubełka (`size`) i zmieniając wynik na liczbę całkowitą.

```
 # Przypisujemy liczbę do odpowiedniego kubełka
for elem in liczby:
    kubelek_indeks = int(elem // size)

# np. int(1.1 // 0.2) = 5
```

Dodajemy liczbę do odpowiedniego kubełka.

```
kubelki[kubelek_indeks].append(elem)
```

W kolejnym kroku sortujemy elementy poszczególnych niepustych kubełków. Elementy wypisujemy w kolejności od najmniejszego do największego. Możemy w tym celu użyć dowolnego stabilnego algorytmu sortującego. W omawianym przykładzie posłużymy się algorytmem sortowania bąbelkowego.

Do przypisania uporządkowanej zawartości kubełka użyjemy standardowej funkcji `enumerate()`, która zwraca dwie wartości: indeks elementu listy i jego wartość. W ten sposób możemy przypisać posortowany kubełek bez konieczności alokowania pamięci dla kolejnej listy kubełków.

```
for ind, val in enumerate(kubelki):

```

```
    kubelki[ind] = sort_babelkowo(val)
```

Wyjaśnijmy najpierw, w jakim celu dzielimy elementy na kubełki, skoro i tak skorzystamy z innego algorytmu sortowania.

Przeanalizujmy sytuację, w której mamy 100 kubełków – w każdym z nich znajduje się 10 elementów, co oznacza, że mamy w sumie 1000 elementów do posortowania.

Gdybyśmy do posortowania tych elementów zastosowali jedynie sortowanie bąbelkowe, algorytm miałby złożoność czasową rzędu n^2, czyli 1000^2, co jest równe 1 000 000.

W rozwiązaniu polegającym na osobnym sortowaniu każdego z kubełków złożoność czasowa algorytmu sortowania bąbelkowego wyniesie $100 \cdot 10^2$, czyli 10 000.

Porównajmy ze sobą dwie liczby:

$$1000^2 > 100 \cdot 10^2 \tag{1}$$

Możemy zauważyć, że zastosowanie sortowania bąbelkowego w tym przypadku wymaga większej liczby operacji.

Należy jednak pamiętać, że nie zawsze liczba elementów w każdym z kubełków będzie taka sama.

Inicjalizujemy listę `posortowane_liczby`, którą wykorzystamy do zapisania wyniku.

```
# Inicjalizujemy zmienną do przechowywania wyniku
posortowane_liczby = []
```

Iterujemy po zawartości każdego z kubełków i dodajemy liczby do wyniku.

```
# Przeglądamy kubełki i dodajemy liczby do wyniku
for kubelek in kubelki:
    for elem in kubelek:
        posortowane_liczby.append(elem)
    return posortowane_liczby
```

Wypisujemy posortowaną listę.

```
```

```
print(sort_kubelk(liczby, n))
```

> **Przykład 1**
>
> Wykonajmy program, w którym pokażemy zawartość kubełków podczas sortowania. Do sortowania zawartości kubełka użyjemy standardowej funkcji języka Python `sorted()` oraz funkcji `enumerate()`, która zwraca dwie wartości: indeks elementu listy i jego wartość.
>
> Wbudowana funkcja `sorted()` wykorzystuje algorytm Timsort. Jest to algorytm hybrydowy, który dla dostatecznie małych zestawów danych przełącza się z algorytmu sortowania przez scalanie na algorytm sortowania przez wstawianie.

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

def sort_kubelk(liczby, n):
    n = len(liczby)
    max_elem = liczby[0]
    for i in range(1, n):
        if liczby[i] > max_elem:
            max_elem = liczby[i]
    # Tworzymy listę kubełków o rozmiarze n+1, aby zapobiec błędo
    kubelki = [[] for _ in range(n+1)]
    # obliczamy rozmiar kubełka, dzielimy największą wartość prze
    size = max_elem / n
    # Przypisujemy liczbę do odpowiedniego kubełka
    for elem in liczby:
        kubelek_indeks = int(elem // size) # wykonamy dzielenie b
        kubelki[kubelek_indeks].append(elem)
    # Sortujemy każdy kubełek za pomocą sortowania bąbelkowego
    # oraz funkcji enumerate
    for ind, val in enumerate(kubelki):
        kubelki[ind] = sort_babelkowo(val)
    # Inicjalizujemy zmienną do przechowywania wyniku
    posortowane_liczby = []
```

```
    # Przeglądamy kubełki i dodajemy liczby do wyniku
    for kubelek in kubelki:
        for elem in kubelek:
            posortowane_liczby.append(elem)
    return posortowane_liczby

liczby = [0.1, 0.4, 0.2, 0.9, 0.5, 0.8, 0.3, 0.7, 0.6]
n = len(liczby)
print(sort_kubelk(liczby, n))

# efekt
# [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

# dodatkowo uruchomimy kod, podając 3 kubełki
print(sort_kubelk(liczby, 3))

# efekt
# [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

liczby = [0.1, 0.4, 9.345, 0.93, 0.2, 0.9, 1.5, 0.8, 2.3, 0.7, 0.
print(sort_kubelk(liczby, 3))

# efekt
# [0.1, 0.2, 0.4, 0.6, 0.7, 0.8, 0.9, 0.93, 1.5, 2.3, 3, 3.22, 3.
```

Ciekawostka

Możemy zauważyć, jak zmienia się rozkład kubełków, kiedy dodamy wartość mocno odbiegającą od reszty danych, np. `3.453452345234`.

```
liczby = [0.7979395529003974, 3.453452345234, 0.062238742532034
print(sort_kubelk_print(liczby,10))

# efekt:
# Kubełek: [0.00773510734757199, 0.05430109642091918, 0.0622387
# Kubełek: [0.40327316514507294, 0.4121043216002158, 0.43407714
# Kubełek: [0.7134526678493234, 0.7221481031340844, 0.757526365
# Kubełek: []
# Kubełek: []
# Kubełek: []
# Kubełek: []
```

```
# Kubełek: []
# Kubełek: []
# Kubełek: []
# Kubełek: [3.453452345234]
# [0.00773510734757199, 0.05430109642091918, 0.0622387425320346
```

Słownik

indeks listy

numer, dzięki któremu możemy odnieść się do konkretnego elementu w tej liście

kubełek

komórka w pamięci, obiekt, czy innego rodzaju struktura, identyfikowana jednoznacznie poprzez unikalną cechę sortowanego zbioru (w przypadku listy liczb całkowitych każdy kubełek identyfikowany byłby unikalną liczbą), w których zliczamy liczbę wystąpień elementów, zawierających daną cechę

złożoność czasowa

ilość czasu potrzebnego do wykonania zadania, wyrażona jako funkcja ilości danych

złożoność obliczeniowa

ilość zasobów komputerowych potrzebnych do wykonania zadania

Symulacja interaktywna

Polecenie 1

Zapoznaj się z symulacją przedstawiającą sortowanie kubełkowe losowo wygenerowanych liczb. Symulacja umożliwia zmianę liczby sortowanych elementów, a także liczbę kubełków. Zaobserwuj przebieg procesu sortowania kubełkowego.

Zapoznaj się z symulacją przedstawiającą sortowanie kubełkowe losowo wygenerowanych liczb. Symulacja umożliwia zmianę liczby sortowanych elementów, a także liczbę kubełków. Zaobserwuj przebieg procesu sortowania kubełkowego.

Ważne!

W tej symulacji elementy w kubełku są sortowane na bieżąco w momencie, kiedy trafiają do niego. Algorytm zakłada, że sortowanie elementów wewnątrz kubełka odbywa się dopiero wtedy, kiedy wszystkie elementy będą już umieszczone w kubełkach.

Polecenie 2

Oblicz, jaka byłaby złożoność czasowa tego algorytmu, gdyby do sortowania elementów wewnątrz kubełków wykorzystać wyłącznie **sortowanie przez wstawianie**.

Polecenie 3

Oblicz, jaka byłaby złożoność czasowa tego algorytmu, gdyby do sortowania elementów wewnątrz kubełków wykorzystać wyłącznie **sortowanie bąbelkowe**.

Polecenie 4

Zastanów się, w jakich sytuacjach życia codziennego algorytm sortowania wykorzystujący kubełki mógłby mieć zastosowanie.

Sprawdź się

Pokaż ćwiczenia: 

Ćwiczenie 1

Dokończ program wyszukujący wśród elementów listy wartość najmniejszą i największą.

Swoje rozwiązanie przetestuj dla następującej listy:

```
tab = [76, 84, 79, 60, -75, 58, 94, 53, -84, 51, -4, -41, 49, -
```

Specyfikacja problemu:

Dane:

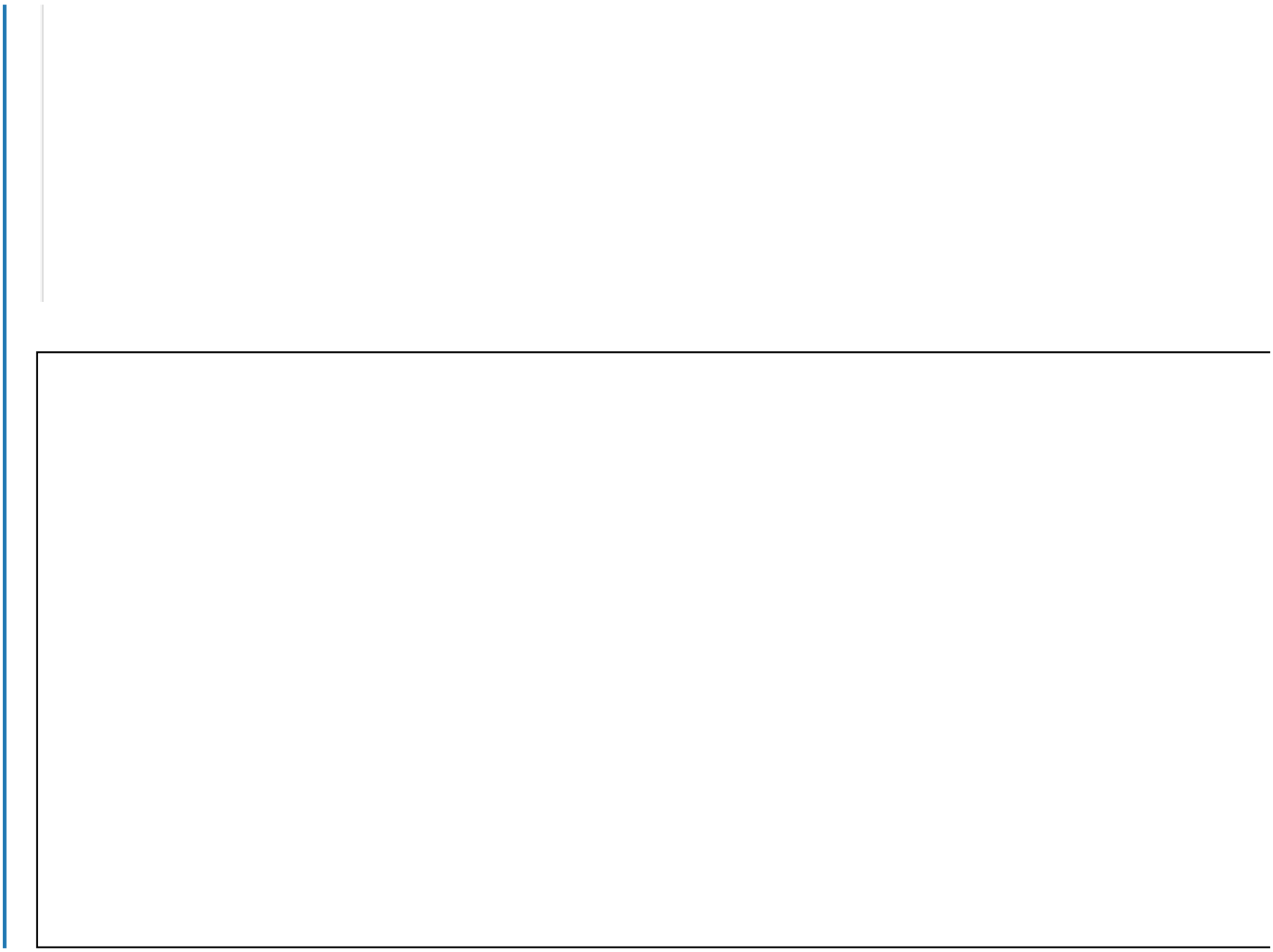
- `tab` – lista zawierająca liczby całkowite z przedziału $[-99,\ 99]$

Wynik:

- `min_elem`, `max_elem` – liczby całkowite; najmniejsza oraz największa wartość spośród liczb zapisanych w liście `tab`

Twoje zadania

1. Wypisz największą liczbę z listy.
2. Wypisz najmniejszą liczbę z listy.



Ćwiczenie 2

Napisz program, który dla zadanej listy zliczy wystąpienia każdego z elementów do odpowiedniego kubełka, a następnie wypisze zawartość listy z kubełkami.

Swoje rozwiązanie przetestuj dla następującej listy:

```
tab = [76, 84, 79, 60, -75, 58, 94, 53, -84, 51, -4, -41, 49, -
```

Specyfikacja problemu:

Dane:

- `tab` – lista zawierająca liczby całkowite z przedziału $[-99, 99]$

Wynik:

- `kubelki` – lista liczb całkowitych; zawartość listy z kubełkami

Twoje zadania

1. Poprawnie wypisz zawartość kubełków.

Ćwiczenie 3

Zaimplementuj algorytm sortowania kubełkowego i wypisz wynik sortowania. Listę posortuj od największego do najmniejszego elementu.

Swoje rozwiązanie przetestuj dla następującej listy:

```
tab = [76, 84, 79, 60, -75, 58, 94, 53, -84, 51, -4, -41, 49, -
```

Specyfikacja problemu:

Dane:

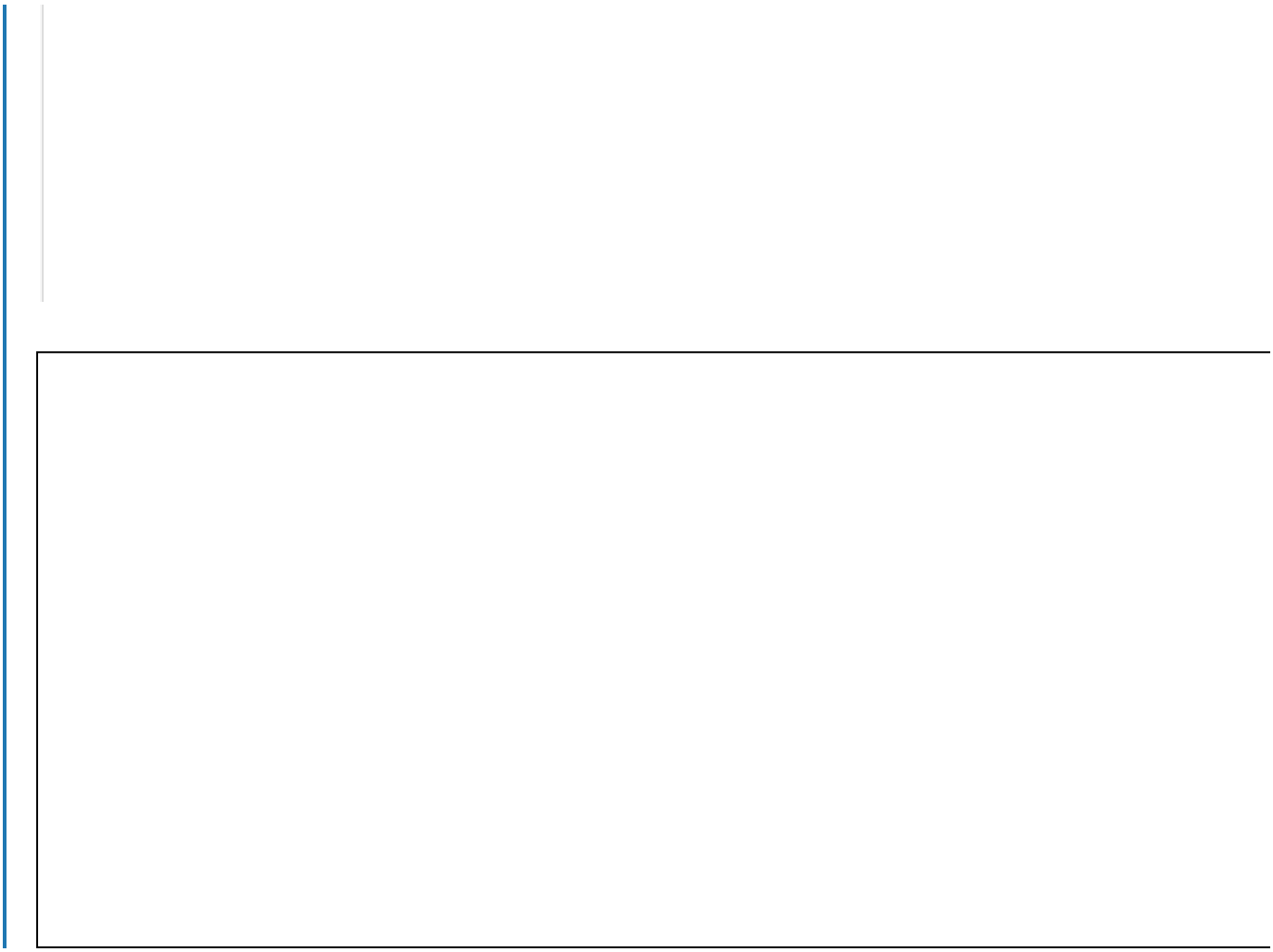
- `tab` – lista zawierająca liczby całkowite z przedziału $[-99,\ 99]$

Wynik:

- `posortowane_elem` – lista zawierająca liczby całkowite z przedziału $[-99,\ 99]$ posortowane nierosnąco

Twoje zadania

1. Posortuj listę nierosnąco.



Ćwiczenie 4

Wykorzystując algorytm sortowania kubełkowego, zapisz program, który zwróci liczbę niewykorzystanych kubełków podczas etapu podziału danych.

Napisz funkcję `ile_pustych()`, która zwróci odpowiednie dane.

Przykładowa zawartość listy kubełków dla następujących danych wejściowych:

```
liczby = [2.88, 1.94, 1.36, 1.8, 5.28, 3.07, 1.11, 4.59, 5.76,
print(ile_pustych(liczby, 6))
# efekt
# 1
```

Specyfikacja problemu:

Dane:

- `liczby` – lista zawierająca nieposortowane liczby rzeczywiste dodatnie
- n – liczba kubełków

Wynik:

- `puste` – liczba naturalna `int`; liczba pustych, niewykorzystanych kubełków

Twoje zadania

1. Program zwraca liczbę niewykorzystanych w czasie sortowania danych kubełków.

Dla nauczyciela

Autor: Adam Jurkiewicz
Przedmiot: Informatyka

Temat: Sortowanie kubełkowe w języku Python

Grupa docelowa:

Liceum ogólnokształcące i technikum, liceum ogólnokształcące, technikum, zakres rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) objaśnia dobrany algorytm, uzasadnia poprawność rozwiązania na wybranych przykładach danych i ocenia jego efektywność;

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) sprawnie posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

I + II. Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

1) zapisuje za pomocą listy kroków, schematu blokowego lub pseudokodu, i implementuje w wybranym języku programowania, algorytmy poznane na wcześniejszych etapach oraz algorytmy:

d) jednoczesnego wyszukiwania elementu najmniejszego i największego,

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Przeanalizujesz implementację algorytmu sortowania kubełkowego liczb rzeczywistych oraz całkowitych w języku Python.
- Zaimplementujesz algorytm sortowania kubełkowego w języku Python.
- Rozwiążesz ćwiczenia sprawdzające twoją wiedzę dotyczącą algorytmu sortowania kubełkowego.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- oprogramowanie dla języka Python 3 (lub nowszej wersji), w tym PyCharm lub IDLE.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Sortowanie kubełkowe w języku Python". Uczniowie mają zapoznać się z treściami w sekcji „Przeczytaj".

Faza wstępna:

1. Nauczyciel prosi wybraną osobę o odczytanie tematu lekcji, a następnie określa cele i kryteria sukcesu.
2. **Rozpoznanie wiedzy uczniów.** Uczniowie tworzą pytania dotyczące tematu zajęć, na które odpowiedzą w trakcie lekcji.

Faza realizacyjna:

1. **Praca z tekstem.** Nauczyciel ocenia, na podstawie informacji na platformie, stan przygotowania uczniów do zajęć. Jeżeli jest ono niewystarczające, prosi wybraną osobę o przedstawienie najważniejszych informacji z sekcji „Przeczytaj".
   Uczniowie analizują przykład z sekcji „Przeczytaj". Następnie w parach powtarzają i testują zaprezentowane rozwiązanie na swoim komputerze.
2. **Praca z multimedium**. Nauczyciel wyświetla zawartość sekcji „Symulacja interaktywna". Uczniowie zapoznają się z poleceniem nr 1. Wykonują je indywidualnie, następnie omawiają swoje spostrzeżenia na forum klasy.
3. **Ćwiczenie umiejętności.** Uczniowie wykonują indywidualnie ćwiczenie nr 1 z sekcji „Sprawdź się", a następnie porównują swoje odpowiedzi z kolegą lub koleżanką.
4. W kolejnym etapie uczniowie dobierają się w pary i wykonują ćwiczenia nr 2-3 z sekcji „Sprawdź się" . Następnie konsultują swoje rozwiązania z inną parą uczniów i ustalają jedną wersję odpowiedzi.

Faza podsumowująca:

1. Na koniec zajęć nauczyciel raz jeszcze wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie". W odniesieniu do ich realizacji dokonuje szczegółowej oceny rozwiązania zastosowanego przez wybranego ucznia.
2. Wybrany uczeń podsumowuje zajęcia z programowania w Pythonie, zwracając uwagę na nabyte umiejętności.

Praca domowa:

1. Uczniowie wykonują polecenia 2-4 z sekcji „Symulacja interaktywna".
2. Uczniowie wykonują ćwiczenie 4 z sekcji „Sprawdź się".

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka Python 3 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania PyCharm lub IDLE.

Wskazówki metodyczne:

- Uczniowie mogą wykorzystać multimedium w sekcji „Symulacja interaktywna” do przygotowania się do lekcji powtórkowej.