Zestaw 7

Celem zestawu jest studium różnych algorytmów sortowania. Sortowanie to ważna część algorytmiki, zapewne powtarzana wiele razy na różnych przedmiotach, ale warto spróbować również w Pythonie. Materiał będzie prawdopodobnie omówiony na wykładzie, jest również mnóstwo informacji w sieci (strona: https://ufkapano.github.io/algorytmy/lekcja16/index.html). Bardzo zachęcam, aby nie kopiować znalezionych rozwiązań (cudzego gotowego kodu), tylko poświęcić czas na zrozumienie strategii poszczególnych algorytmów oraz je zapisać.

Aby uatrakcyjnić analizę działania poszczególnych algorytmów, przygotowałem (na bazie kodu znalezionego w jednym z „samouczków") uproszczoną, acz efektowną wersję kodu, który tworzy wejściową tablicę z wartościami do posortowania oraz wykonuje animowaną wizualizację procesu sortowania w postaci zmieniającego się dynamicznie histogramu. Użyte są tu biblioteki numpy oraz `matplotlib`, do wykonania animacji użyta zostanie funkcja `FuncAnimation`.

Obliczenia numeryczne i wizualizacje w Pythonie często oparte są na kanonie bibliotek:

- NumPy (https://numpy.org/)
- SciPy (https://scipy.org/)
- Matplotlib (https://matplotlib.org/)

Jeżeli nie mamy ich zainstalowanych, możemy zainstalować je po kolei za pomocą `pip install <nazwa>`, dodatkowo zainstalowane będą wtedy wymagane pakiety zależne. W tym dokumencie zamieszczam tylko nieco komentarzy pomocnych do zrozumienia proponowanego kodu.

Podstawową zaletą biblioteki NumPy jest zaimplementowany w niej typ tablicy (`ndarray`). Do naszych celów (sortowania) potrzebna będzie tablica jednowymiarowa, o jednorodnym rozkładzie indeksów oraz różnej liczbie wartości, rozłożonych losowo lub według jakiegoś uporządkowania. Po zaimportowaniu modułu, do stworzenia tablicy używamy funkcji `linspace`, podając zakres od-do:

```
>>> import numpy as np
>>> np.linspace(0,1000)
array([   0.        ,   20.40816327,   40.81632653,   61.2244898 ,
         81.63265306,  102.04081633,  122.44897959,  142.85714286,
        163.26530612,  183.67346939,  204.08163265,  224.48979592,
        244.89795918,  265.30612245,  285.71428571,  306.12244898,
        326.53061224,  346.93877551,  367.34693878,  387.75510204,
        408.16326531,  428.57142857,  448.97959184,  469.3877551 ,
        489.79591837,  510.20408163,  530.6122449 ,  551.02040816,
        571.42857143,  591.83673469,  612.24489796,  632.65306122,
        653.06122449,  673.46938776,  693.87755102,  714.28571429,
        734.69387755,  755.10204082,  775.51020408,  795.91836735,
        816.32653061,  836.73469388,  857.14285714,  877.55102041,
        897.95918367,  918.36734694,  938.7755102 ,  959.18367347,
        979.59183673, 1000.        ])
```

W ten sposób powstał obiekt tablicy `ndarray`, a domyślna liczba podziałów wynosi 50 i można ją określić jako trzeci parametr (pozycyjny, lub nazwany `num`):

```
>>> np.linspace(0,1000,11) # lub np.linspace(0,1000,num=11)
array([   0.,  100.,  200.,  300.,  400.,  500.,  600.,  700.,  800.,
        900., 1000.])
```

Za pomocą parametru `dtype` można określić typ generowanych elementów, warto pamiętać, że NumPy używa swoje własne typy, na przykład `float64` lub `int64`. Powyższy przykład możemy zapisać:

```
>>> np.linspace(0,1000,11,dtype=np.int64)
array([   0,  100,  200,  300,  400,  500,  600,  700,  800,  900, 1000],
      dtype=int64)
```

Odwrotną kolejność uzyskami zamieniając pierwsze dwa argumenty (można je podawać również jako argumenty nazwane, `start`, `stop`):

```
>>> np.linspace(1000,0,11,dtype=np.int64)
array([1000,  900,  800,  700,  600,  500,  400,  300,  200,  100,    0],
      dtype=int64)
```

Jeśli chcemy wymieszać kolejność tych wartości, możemy to zrobić za pomocą `random.shuffle(tablica)`:

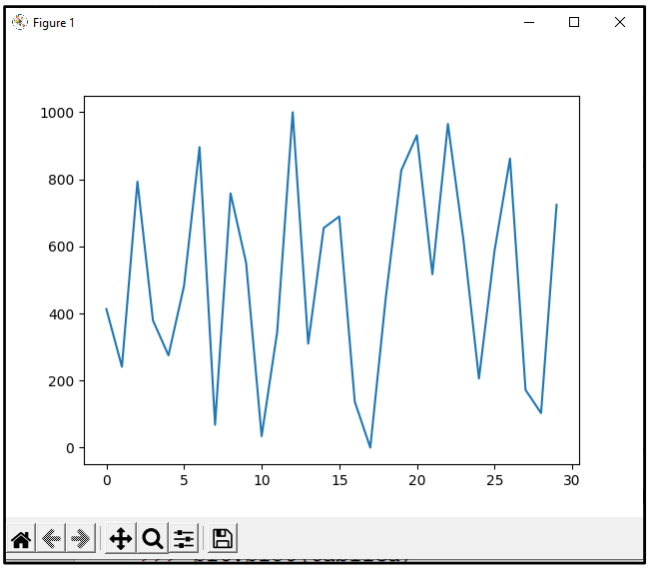
```
>>> tablica = np.linspace(0,1000,30,dtype=np.int64)
>>> tablica
array([   0,   34,   68,  103,  137,  172,  206,  241,  275,  310,  344,
        379,  413,  448,  482,  517,  551,  586,  620,  655,  689,  724,
        758,  793,  827,  862,  896,  931,  965, 1000], dtype=int64)
>>> np.random.shuffle(tablica)
>>> tablica
array([ 413,  241,  793,  379,  275,  482,  896,   68,  758,  551,   34,
        344, 1000,  310,  655,  689,  137,    0,  448,  827,  931,  517,
        965,  620,  206,  586,  862,  172,  103,  724], dtype=int64)
```

W ten sposób generowane są zestawy liczb do badania algorytmów sortowania.

Do ich prezentacji posłuży submoduł `pyplot` z modułu `matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(tablica)
[<matplotlib.lines.Line2D object at 0x00000229E719BF70>]
>>> plt.show()
```

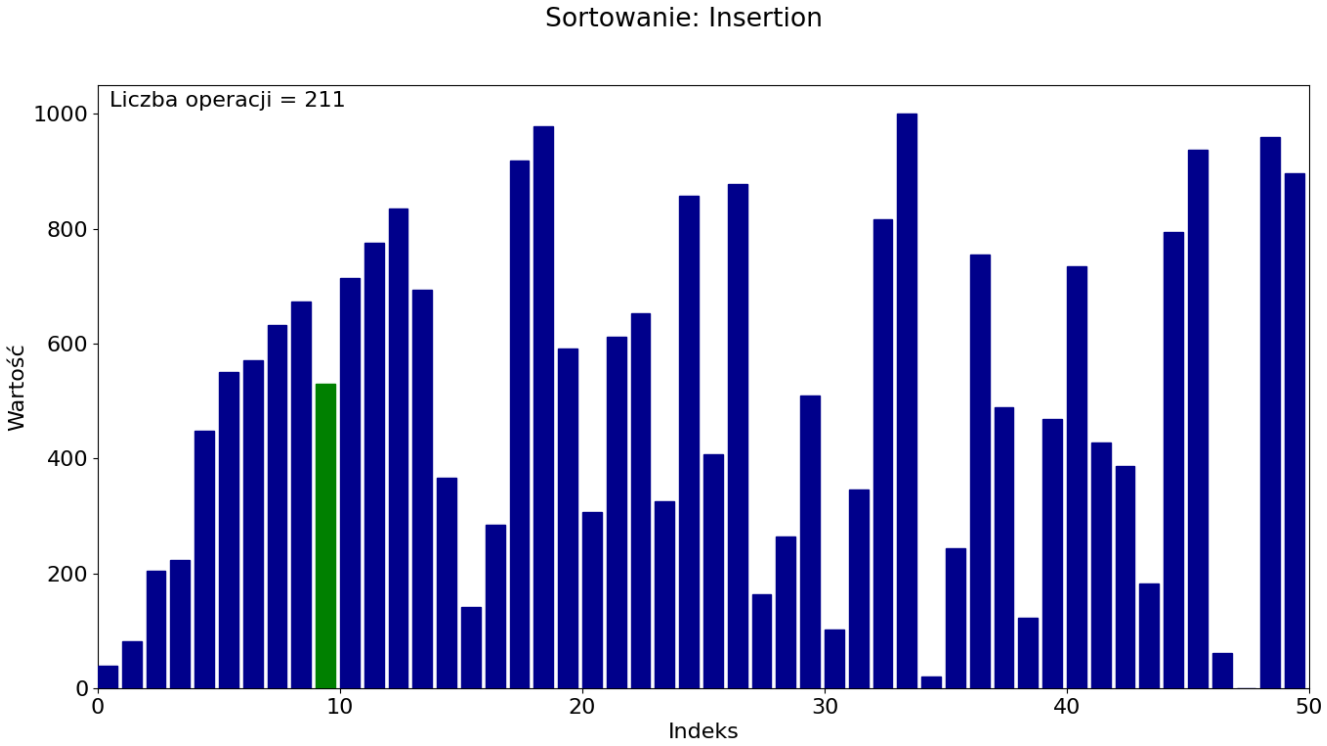
Otrzymamy rysunek:



Wymaga on oczywiście pewnej „obróbki”, ale szczegóły nie są w tym momencie istotne.

Omówię teraz strukturę przygotowanego kodu, na bazie którego należy testować oraz wizualizować działanie poszczególnych algorytmów sortowania, jak w zadaniach poniżej.

Kod składa się, dla zachowania większej czytelności, z dwóch plików. W pliku *mtablica.py* zdefiniowana jest pomocnicza klasa o nazwie `MonitorowanaTablica`, w jej funkcji `__init__` tworzone są, w zależności od trybu, różne tablice: "R" to wartości losowo ułożone, "S" to tablica już posortowana, "A" to tablica posortowana w odwrotnej kolejności, "T" to tablica trzech sekwencji posortowanych (czyli można powiedzieć, posortowana fragmentami). Należy, dla każdego algorytmu, uruchomić każdą z opcji i zobaczyć ile czasu oraz operacji było potrzebne do posortowania. Pozostałe szczegóły implementacji nie są konieczne do wykonania zadania, więc ich tu nie opisuję. Główny plik to *sort1.py*, który importuje potrzebne moduły oraz opisany wyżej typ tablicy. W pliku znajduje się kompletny kod z przykładowym algorytmem sortowania przez wstawianie. Po uruchomieniu, powinniśmy po chwili zobaczyć animację sortowania:



a także wynik, na przykład:

```
Sortowanie: Insertion
Tablica posortowana w czasie 18.4 ms. Liczba operacji: 3730.
```

Kod od linii 32 to szczegóły implementacji sposobu wyświetlania histogramu (np. funkcja `update` steruje kolorami poszczególnych słupków, operacja czytania zamienia słupek na zielony, a zapisu na czerwony), na samym końcu wywołana jest funkcja `FuncAnimation`, która akumuluje kolejne klatki w zmiennej `ani`, a całość jest wyświetlona poprzez `plt.show()`. Wynik animacji można też zapisać jako obrazki poszczególnych klatek i połączyć w film za pomocą jakiegoś programu.

Państwa zadaniem będzie implementacja różnych algorytmów sortowania, których kod należy umieścić w miejscu przykładowego kodu „Insertion Sort”. Dany algorytm można oczywiście zapisać w postaci funkcji. Proszę zwrócić uwagę na pomiar czasu – zaczynający się od odczytu `t0 = time.perf_counter()`, a kończący się na `delta_t = time.perf_counter() - t0`.

Proszę też dla każdego algorytmu zbadać różne przypadki danych początkowych, które ustawia się w linii:

```
tablica = MonitorowanaTablica(0, 1000, N, "R") # zbadaj też opcje: "S", "A", "T"
```

Zadania

Używając kod z pliku *sort1.py* oraz *mtablica.py*, proszę zaimplementować i przebadać następujące algorytmy sortowania.

1. Bubble sort (https://ufkapano.github.io/algorytmy/lekcja16/bubblesort.html)
2. Shell sort (np. wg Sedgewicka: https://ufkapano.github.io/algorytmy/lekcja16/shellsort.html)
3. Merge sort (https://ufkapano.github.io/algorytmy/lekcja16/mergesort.html)
4. Quick sort (https://ufkapano.github.io/algorytmy/lekcja16/quicksort.html)
5. Tim sort (https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python)

Proszę, **oprócz kodu algorytmów**, koniecznie zapisać i wysłać **plik z wynikami** pomiarowymi. Może to być prosty plik ASCII, o takiej zawartości:

Insert sort
R: Tablica posortowana w czasie 18.4 ms. Liczba operacji: 3730.
S: Tablica posortowana w czasie 0.6 ms. Liczba operacji: 98.
A: Tablica posortowana w czasie 81.2 ms. Liczba operacji: 7350.
T: Tablica posortowana w czasie 11.1 ms. Liczba operacji: 2254.

Bubble sort
R:
S:
A:
T:

i tak dalej.