## Zestaw 5

**1.** Celem tego zadania jest studium wybranych 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`.

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)`:

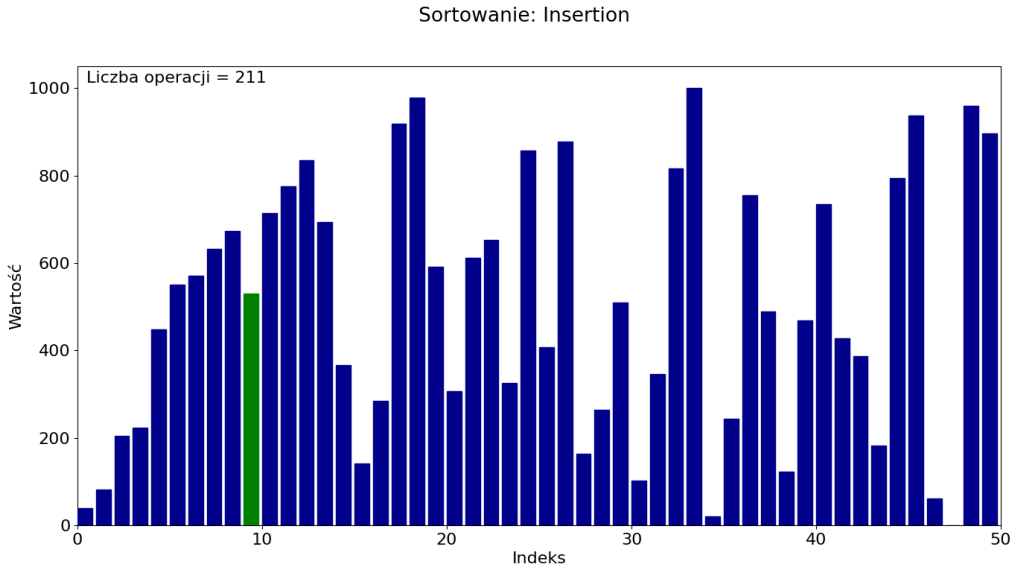
```
>>> tablica = np.linspace(0,1000,30,dtype=np.int64)
>>> tablica
array([   0,   34,   68,  103,  137,  172,  206,  241,  275,  310,  344,
        379,  413,  448,  482,  517,  551,  586,  620,  655,  689,  724,
        758,  793,  827,  862,  896,  931,  965, 1000], dtype=int64)
>>> np.random.shuffle(tablica)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

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 kilku plików. W pliku `ZADANIE1/tablica.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 (rosnąco), `"A"` to tablica posortowana w odwrotnej kolejności (czyli malejąco), `"T"` to tablica trzech sekwencji posortowanych (czyli można powiedzieć, posortowana fragmentami). Proszę przyjrzeć tej klasie (`MonitorowanaTablica`) – dzięki przechowywaniu pełnych kopii tablicy po każdej operacji można analizować zmiany w czasie. Wewnętrzne listy pozwalają na monitorowanie, które indeksy są odczytywane i modyfikowane, co będzie przydatne w analizie algorytmów (np. sortowania), aby zrozumieć wzorce dostępu do danych. Metody `__getitem__` i `__setitem__` pozwalają definiować, jak obiekt klasy zachowuje się, gdy korzystamy z operatorów indeksowania `[]`.

W pliku `ZADANIE1/histogram.py` znajduje się funkcja `plot_histogram`, która wizualizuje proces sortowania za pomocą animacji histogramu, co pozwala zobaczyć, jak zmienia się stan tablicy w kolejnych krokach algorytmu. Najpierw ma miejsce inicjalizacja (rozmiar czcionki, wykres i jego osie) i narysowanie pierwszego histogramu `ax.bar(…)` (paski reprezentujące elementy tablicy), bazując na pierwszym stanie tablicy (`tablica.pelne_kopie[0]`). Metoda `ax.bar(…)` w Matplotlib służy do tworzenia wykresów słupkowych (histogramów), gdzie prostokąty (słupki) reprezentują wartości danych. Następnie tytuł wykresu z nazwą sortowania itd., `txt = ax.text(…)` dodaje tekst w rogu wykresu, który będzie aktualizowany w trakcie animacji (liczba operacji sortowania). Funkcja aktualizująca histogram (`update`), również liczbę wykonanych operacji, dostosowuje wysokość każdego słupka histogramu (`rectangle.set_height(height)`) na podstawie aktualnego stanu tablicy w `tablica.pelne_kopie[frame]`. Indeks i typ operacji pobrane są z `tablica.aktywnosc(frame)`, prowadząc do wyświetlenia operacji odczytu (`get`) kolorem zielonym (`green`) i zapisu (`set`) kolorem czerwonym (`red`). Wreszcie, `FuncAnimation(...)` tworzy animację histogramu. Argumenty: `fig` – wykres, który będzie animowany; `update` – funkcja odpowiedzialna za aktualizację wykresu w każdej klatce, otrzymuje automatycznie argument, indeks klatki; `frames` – liczba klatek w animacji, odpowiada liczbie stanów tablicy zapisanych w `tablica.pelne_kopie`; `blit=True` – optymalizuje animację, aktualizując tylko zmienione elementy; `interval=1000./fps` – ustawia odstęp między klatkami w milisekundach na podstawie liczby klatek na sekundę (`fps`); `repeat=False` – animacja jest odtwarzana tylko raz; `plt.show()` – wyświetla animację na ekranie. Tak naprawdę powyższa wiedza nie jest konieczna do zadania, ale jest pożyteczne zapoznać się z mechanizmem działania animacji. Żeby zobaczyć efekt, **należy odkomentować linię z kodem** `plot_histogram(tablica, algo_pomiar["nazwa"], FPS)` w pliku `main.py`. Wtedy, po uruchomieniu programu, powinniśmy po chwili zobaczyć animację sortowania:




Główny plik `ZADANIE1/main.py` definiuje parametry wejściowe, `N` – rozmiar tablicy (liczba elementów do posortowania); `FPS` – liczba klatek na sekundę dla animacji; `TRYBY` – lista trybów tworzenia tablicy (jak omówiono powyżej). Następnie funkcja `wykonaj_pomiar` obsługuje pomiar czasu wykonania algorytmów sortowania dla różnych trybów tablic i zebranie wyników w formacie nadającym się do analizy. Zwróćmy uwagę, że algorytmy i ich nazwy wczytywane są z listy `algorytmy` (plik `algorytmy.py`) z krotkami zawierającymi nazwę funkcji danego algorytmu sortowania oraz opis – łańcuch znakowy. Funkcja zwraca słownik zawierający dwa elementy - `"nazwa"`: nazwa algorytmu, oraz `"pomiary"` : listę zagnieżdżonych słowników z wpisami `"tryb"`, `"czas"` i `"operacje"`. Wszystko to zostaje zapisane finalnie, poprzez wywołanie metody `zapisz()`, do pliku wyjściowego `"pomiary.txt"`. Zanim przejdziemy do meritum zadania, przedstawiam przykładową zawartość takiego pliku, należy zwrócić uwagę na liczbę wykonanych operacji w przypadku różnych rodzajów sortowania:

```
Insertion Sort
R: Tablica posortowana w czasie 0.5 ms. Liczba operacji: 170.
S: Tablica posortowana w czasie 0.1 ms. Liczba operacji: 18.
A: Tablica posortowana w czasie 4.1 ms. Liczba operacji: 270.
T: Tablica posortowana w czasie 0.5 ms. Liczba operacji: 70.

Bubble Sort
R: Tablica posortowana w czasie 1.3 ms. Liczba operacji: 194.
S: Tablica posortowana w czasie 0.6 ms. Liczba operacji: 18.
A: Tablica posortowana w czasie 2.1 ms. Liczba operacji: 270.
T: Tablica posortowana w czasie 1.6 ms. Liczba operacji: 96.

Shell Sort
R: Tablica posortowana w czasie 1.0 ms. Liczba operacji: 100.
S: Tablica posortowana w czasie 0.3 ms. Liczba operacji: 45.
A: Tablica posortowana w czasie 0.7 ms. Liczba operacji: 77.
T: Tablica posortowana w czasie 0.3 ms. Liczba operacji: 51.

Merge Sort
R: Tablica posortowana w czasie 2.8 ms. Liczba operacji: 112.
S: Tablica posortowana w czasie 0.8 ms. Liczba operacji: 106.
A: Tablica posortowana w czasie 0.8 ms. Liczba operacji: 98.
T: Tablica posortowana w czasie 0.7 ms. Liczba operacji: 98.

Quick Sort
R: Tablica posortowana w czasie 1.2 ms. Liczba operacji: 90.
S: Tablica posortowana w czasie 2.4 ms. Liczba operacji: 270.
A: Tablica posortowana w czasie 2.5 ms. Liczba operacji: 170.
T: Tablica posortowana w czasie 2.0 ms. Liczba operacji: 124.

Tim Sort
R: Tablica posortowana w czasie 1.7 ms. Liczba operacji: 170.
S: Tablica posortowana w czasie 0.2 ms. Liczba operacji: 18.
A: Tablica posortowana w czasie 1.8 ms. Liczba operacji: 270.
T: Tablica posortowana w czasie 0.4 ms. Liczba operacji: 70.
```

Zasadnicza część zadania znajduje się w pliku `ZADANIE1/algorytmy.py`. Żeby oswoić się z kodem oraz sposobem jego działania, trzy z wyżej wymienionych algorytmów:

- Insertion Sort ( https://ufkapano.github.io/algorytmy/lekcja16/insertsort.html )
- Bubble sort ( https://ufkapano.github.io/algorytmy/lekcja16/bubblesort.html )
- Shell sort ( np. wg Sedgewicka: https://ufkapano.github.io/algorytmy/lekcja16/shellsort.html ),

są całkowicie zaimplementowane.

**Wymagania formalne** Uzupełnić pliki `ZADANIE1/algorytmy.py` w repozytorium GitHub Classroom algorytmami:

- Merge sort ( https://ufkapano.github.io/algorytmy/lekcja16/mergesort.html )
- Quick sort ( https://ufkapano.github.io/algorytmy/lekcja16/quicksort.html )
- Tim sort ( https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python ).

W pliku przygotowane są metody o odpowiednich nazwach, a na dole pliku lista `algorytmy`, z zakomentowaną częścią, którą trzeba uzupełnić i odkomentować. Uruchamiając kod z **zakomentowanym** wywołaniem funkcji `plot_histogram()`, powstanie plik wynikowy `"pomiary.txt"`, który **również należy wysłać do repozytorium**! Proszę zachować parametr `N = 50`. Testom poddane zostaną wyniki (liczba operacji) z poszczególnych algorytmów i danych wejściowych. Dopuszczalne jest odchylenie 10% od wartości referencyjnych, zapisanych w pliku z testami. Pierwsze trzy testy dają oczywiście wynik poprawny, bo są na bazie przekazanego kodu [**zadanie za 3 pkt**].

**2.** To zadanie ma nieco inny charakter niż poprzednie, ale oczywiście dotyczy języka Python. Proponuję, aby rozwiązali Państwo dwa wybrane zadania z serwisu https://www.hackerrank.com/. W tym celu należy na powyższej stronie najpierw założyć konto (`Sign Up` & `I'm here to practice and prepare`), po przejściu kilku pytań wybieramy Python: https://www.hackerrank.com/domains/python. Osobliwość rozwiązywania zadań w tego typu serwisach polega na tym, że rozwiązania są poddane intensywnym testom i nierzadko rozwiązanie wyglądające na poprawne, nie przechodzi wszystkich testów. Zamiast pisać treść zadań, poniżej podaję tytuły zadań do rozwiązania. Należy próbować, aż do osiągnięcia pełnego sukcesu! Wybrane zadania to:

- Reduce Function
- Maximize It!

**Wymagania formalne** Uzupełnić pliki `ZADANIE2/reduce_function.py` i `ZADANIE2/maximize_it.py` w repozytorium GitHub Classroom kodem, który przechodzi wszystkie testy w serwisie `hackerrank`, ale również przechodzi testy lokalne. Proszę zauważyć, że w pliku `maximize_it.py` część kodu już jest napisana (w serwisie `hackerrank` nie ma w ogóle), więc należy rozwinąć ten kod [**zadanie za 2 pkt**]. P.S. Oczywiście gorąco polecam przerobić więcej zadań z tego serwisu!