Julia Grzegorzewska, Wiktoria Fimińska
Link do repozytorium: https://github.com/grzesiaaa/Algorytmy_lista4

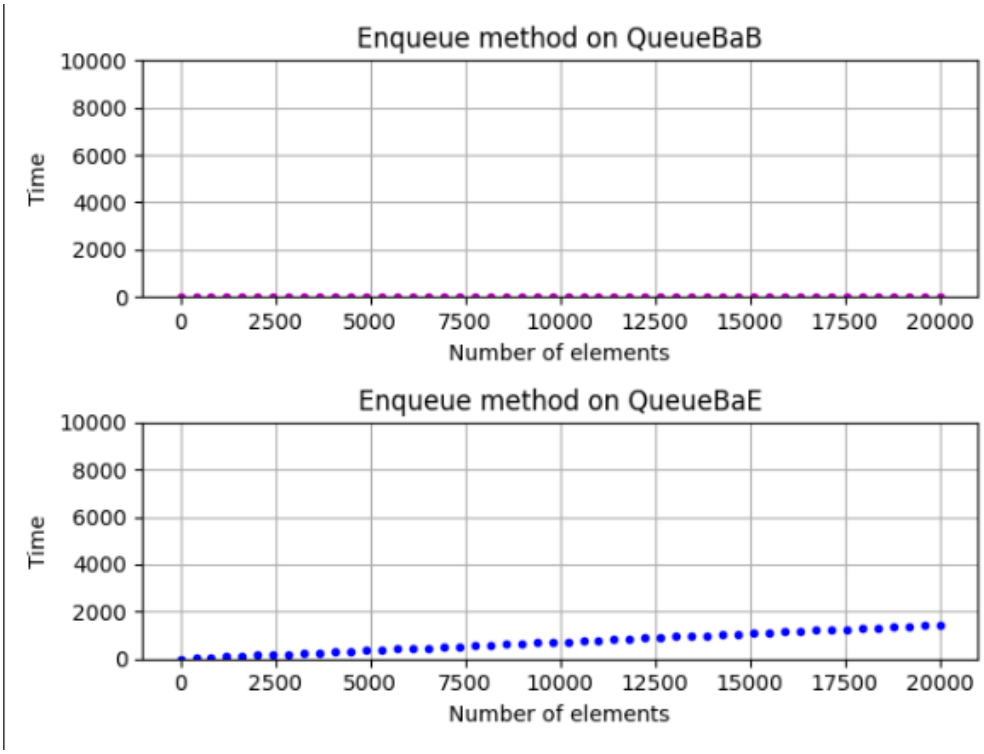
RAPORT LISTA 4

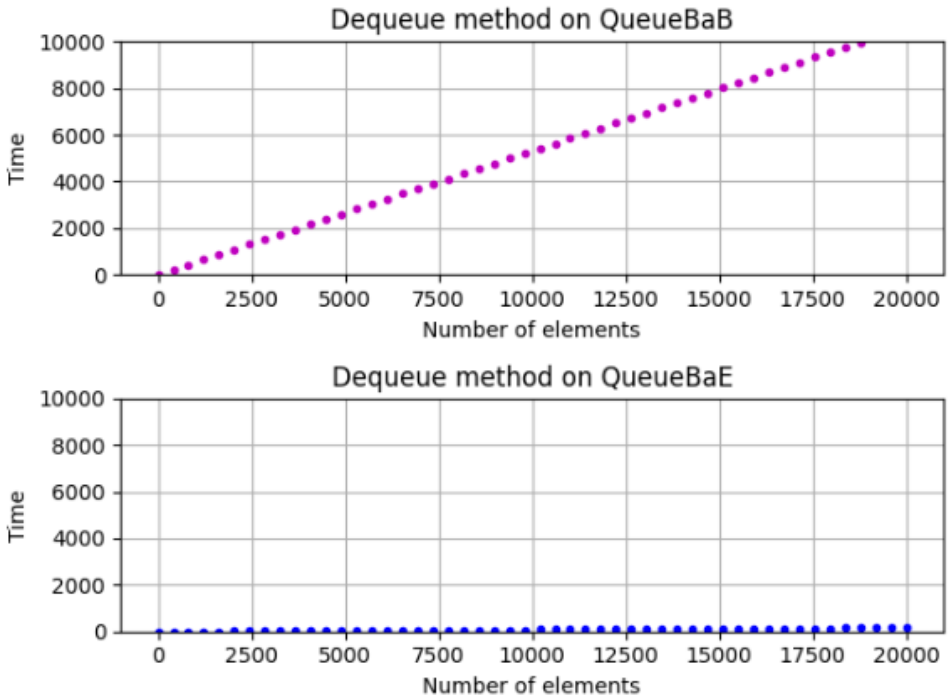
Zadanie 1

Tworzymy kolejkę przy użyciu pythonowych list. W QueueBab początek kolejki przechowujemy na początku listy, więc aby dodać coś do kolejki używamy metody append na liście (dodajemy na koniec), aby ściągnąć – pop (ściągamy ostatni element). W QueueBaE natomiast początek kolejki przechowujemy na końcu listy, więc dodajemy za pomocą metody insert na początek listy (indeks 0), a aby ściągnąć „popujemy" ostatni element (indeks -1). Reszta metod jest taka sama dla obu implementacji (dokładny opis co robi dana metoda znajduje się w kodzie).
W QueueBab na potrzeby zadania 3 dodałyśmy jeszcze metodę first, która zwraca element znajdujący się na początku kolejki.

Zadanie 2

Eksperyment porównujący wydajność obu implementacji kolejek z zadania 1 polega na stworzeniu wykresów przedstawiających czas wykonania metod enqueue i dequeue dla różnej ilości elementów. W funkcji count_times zliczamy te czasy, a następnie w funkcjach compare_enqueue i compare_dequeue tworzymy wykresy z otrzymanych danych.
Prezentują się one następująco dla 20000 elementów:





Widzimy stąd, że implementacja QueueBaB jest bardziej optymalna jeśli chodzi o wstawianie elementów do kolejki, natomiast QueueBaE przy usuwaniu elementów.
Z wykładu:
Dla QueueBaB wstawianie rzędu O(1), usuwanie O(n)
Dla QueueBaE wstawianie rzędu O(n), usuwanie O(1)

Zadanie 3

W zadaniu 3 mieliśmy stworzyć własną symulację z wykorzystaniem kolejki. Jako symulację wybrałyśmy sytuację polegającą na określeniu prędkości wpuszczania ludzi do auli na koncert w zależności od tego czy ludzie byli wpuszczani pojedynczo, czy bilety można było zakupić grupowo. Uczestnicy koncertu otrzymują różne atrybuty.
Symulacje przeprowadzamy w różnych wariantach:
-gdy można zakupić bilety tylko pojedynczo;
-gdy można zakupić zarówno bilety grupowe, jak i pojedyncze;
-gdy można zakupić bilety tylko grupowo.

Klasa *Participant* przyjmuje atrybuty *numer* i *acompany*, które oznaczają kolejno numer uczestnika w kolejce oraz ilość osób, z którymi uczestnik przyszedł.
Klasa *Philharmonic* tworzy klasę, która opisuje ile drzwi jest przy zakupie biletu.

Korzystamy z modułu *numpy.random.choicel*, który pozwala nam z danym prawdopodobieństwem określić iluosobowe bilety zostaną zakupione. Funkcja *list_of_probable_values* tworzy listę o długości *lenght*(odpowiada ilości sprzedanych biletów).

W funkcji *types_of_tickets* tworzymy jedną kolejkę z uczestników i w zależności od listy zakupionych rodzajów biletów i przypisuje atrybuty obiektom z klasy *Participants*. Zostały wydzielone 3 przypadki: *„mixed”, „singular”* oraz *„plural”* (wśród zakupionych biletów są zarówno pojedyncze, jak i grupowe; tylko pojedyncze; tylko grupowe). Zwraca nam listę uczestników.

Następnie przechodzimy do symulacji. Przyjmuje ona wartości takie jak: liczba osób zainteresowanych koncertem, liczba drzwi w filharmonii, typy biletów oraz ilości zakupionych biletów przez poszczególnych uczestnikow. Tworzymy kolejkę osób *participants* oraz w pętli tworzymy kolejne kolejki w zależności od liczby drzwi. Kolejkę *participants* dzielimy na mniejsze kolejki i tworzymy z nich listę. Kolejno przechodzimy przez każdą z nich i dopóki nie jest pusta, przekazujemy wartości z ilości zakupionycy biletów i tworzy listę jak długo trwa proces zakupu oraz wejścia do filharmonii. Na koniec sumuje wartości czasów i zwraca je.

Nasza teza to: bilety sprzedawane grupowo będą sprzedawane szybciej.

Zadanie 4

Aby sprawdzić poprawność składni dokumentu HTML posłużymy się stosem (klasa Stack wzięta z wykładu). Najpierw wczytujemy plik i pozbywamy się z niego wszystkich komentarzy oraz robimy z jego znaków listę. Tworzymy też listę znaczników, które nie wymagają zamknięcia, żeby nie sprawdzać ich poprawności pod tym kątem. Lecimy pętlą po wczytanym pliku i tworzymy stringa z tych znaków, które znajdują się między znakiem „<" i „>" a następnie dodajemy go do naszej listy tagów (bierzemy tylko pierwszy wyraz, omijamy to co jest w tagu po spacji). Potem tworzymy nową listę, w której pomijamy te znaczniki nie wymagające zamknięcia. Mając już tę listę zaczynamy sprawdzać poprawność tych znaczników. Tzn. dodajemy na stos znacznik, potem jeśli kolejny rózni się jedynie znakiem „/" to ściągamy wierzchni element stosu, a jeśli nie to dodajemy ten znacznik na stos i tak aż do końca listy. Jeśli na koniec stos będzie pusty zwracamy True (kod poprawny), jeśli nie False.

Zadanie 5

Dodajemy brakujące metody append, insert, index i pop do istniejącej już klasy UnorderedList z wykładu.

Append - dodanie elementu na koniec listy
Jeśli chcemy coś dodać na koniec pustej listy to używamy metody add.
W innym przypadku zaczynamy od pierwszego elementu czyli self.head. W pętli przechodzimy po kolejnych elementach listy, aż dojdziemy do ostatniego (przerywamy, gdy dojdziemy do None). Jak już jesteśmy na końcu to tworzymy następny element i przypisujemy mu wartość podaną przez użytkownika.

Index – zwraca indeks podanego elementu
Zaczynamy od pierwszego elementu czyli self.head. Ustawiamy zmienną found na False i indeks na 0. W pętli sprawdzamy, czy dany element jest równy temu, który chcemy znaleźć – jeśli nie to przechodzimy do kolejnego, tym samym zwiększając indeks o 1 (i tak aż do końca listy), a jeśli tak to zmieniamy found na True po czym zwracamy aktualny indeks. Jeśli nie uda nam się znaleźć pasującego elementu to zwracamy None.

Insert – umieszcza we wskazanym miejscu listy podany element
Zaczynamy od pierwszego elementu czyli self.head, więc wcześniejszy element to None. Ustawiamy indeks na 0. Jeśli użytkownik poda pozycję, która nie występuje na liście to wyrzuca IndexError. Jeśli poda pozycję ujemną to zamieniamy ją na odpowiednią pozycję dodatnią. W pętli przechodzimy po kolejnych elementach listy, aż do miejsca, gdzie chcemy wstawić ten podany przez użytkownika (previous zamieniamy na current, current na next, zwiększamy indeks).

Załóżmy, że jesteśmy już w tym docelowym miejscu (current). Chemy między current a previous wcisnąć podany item, więc jako następny po previous ustawiamy item, a jako następny po item ustawiamy current. Jeśli chcemy wstawić item na początek listy to ustawiamy, że następny po item to będzie self.head a self.head zamieniamy na item.

Pop - usuń element z listy na zadanej pozycji
Zaczynamy od pierwszego elementu czyli self.head (wcześniejszy element to None). Ustawiamy indeks na 0. Jeśli lista jest pusta albo użytkownik poda pozycję, która nie występuje na liście to wyrzuca IndexError. Jeśli poda pozycję ujemną to zamieniamy ją na odpowiednią pozycję dodatnią. W pętli przechodzimy po kolejnych elementach listy, aż do elementu, który chcemy usunąć. Czyli mamy w tym momencie nasze elementy previous i current i chcemy pozbyć się current, czyli za następny po previous ustawiamy następny po current i zwracamy element current, by wyświetlić, którego się pozbyliśmy. Gdy chcemy usunąć pierwszy element to ustawiamy, że self.head to następny po current, który był self.headem, a gdy dodatkowo lista ma tylko jeden element to self.head staje się None.

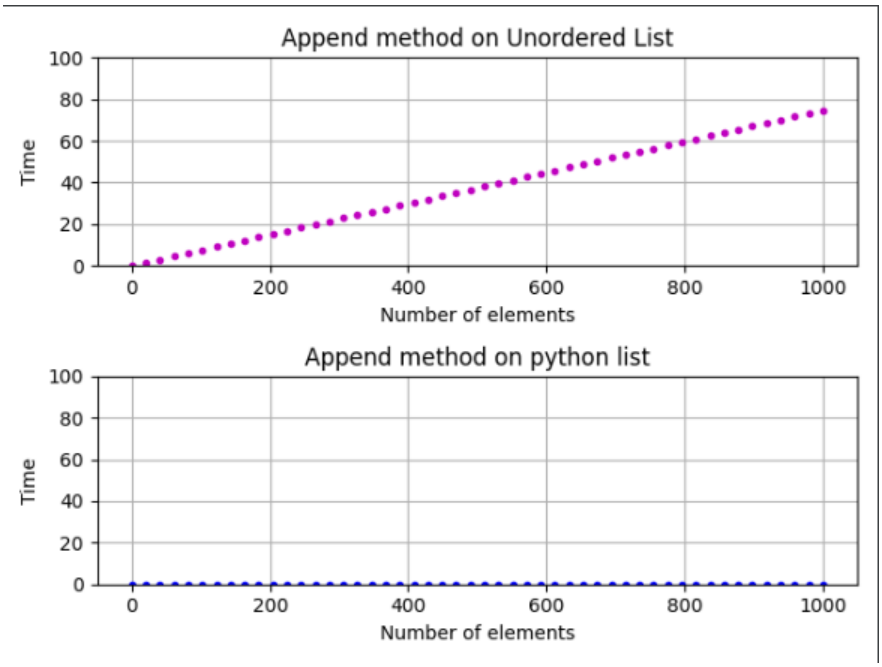
Peek – zwraca ostatni element na liscie
Dodałysmy tę metodę, aby jej użyć w zadaniu 6. Działamy podobnie jak w append jednak na końcu nie dodajemy nowego elementu, ale po prostu wyświetlamy ten ostatni.

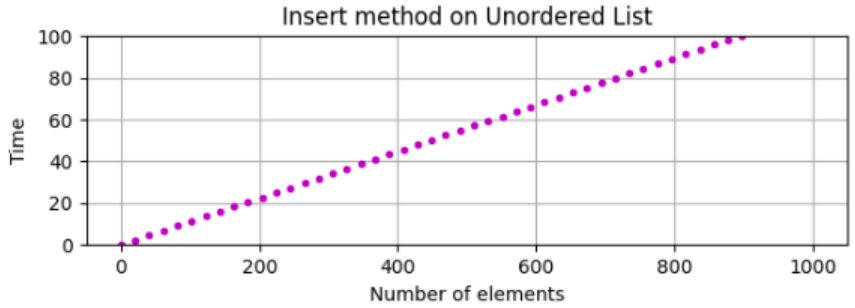
Zadanie 6, 7

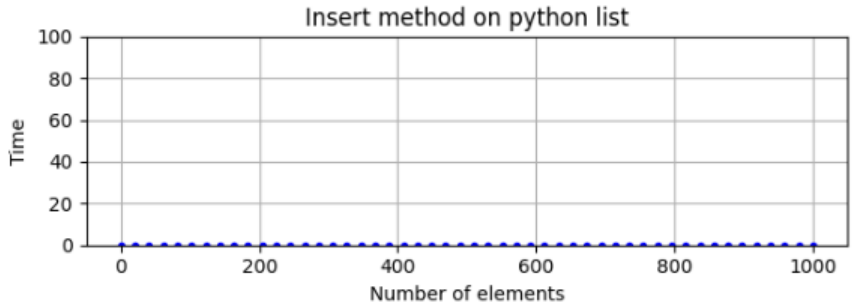
Zadania te polegają na uzyciu odpowiedniej metody z zadania 5.

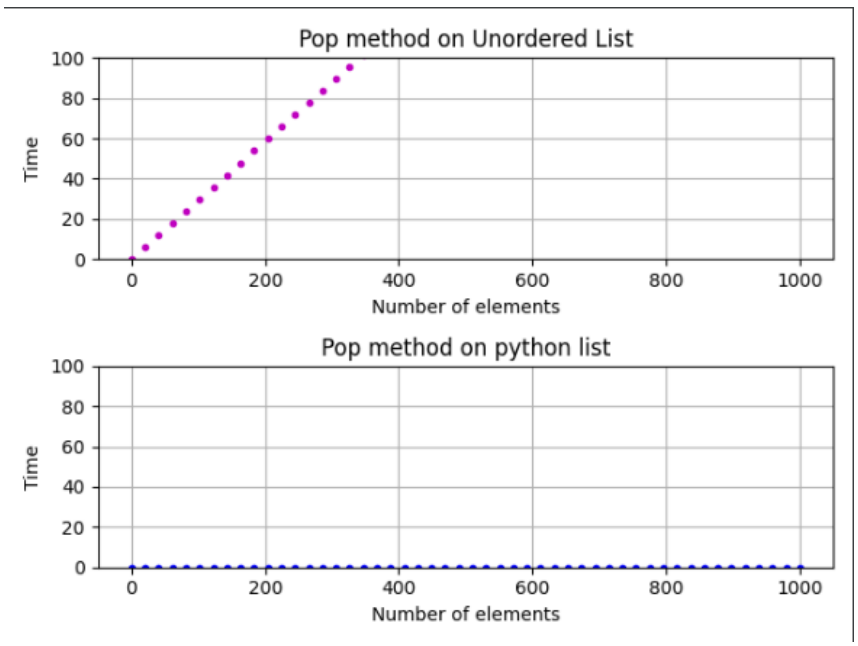
Zadanie 8

Postępujemy podobnie jak w zadaniu 2, czyli tworzymy wykresy przedstawiające czas wykonania różnych operacji dla listy wbudowanej i jednokierunkowej. Funkcja make_plots jest ogólną funkcją do tworzenia wykresów z wygenerowanych danych/czasów i ma na celu uniknięcie powtarzania kodu. Dla każdej porównywanej metody (append, insert,pop) zliczamy czas jej wykonania w zależności od liczby elementów i rodzaju listy a następnie tworzymy wykresy tych czasów.
Oto rezultaty dla n = 1000:

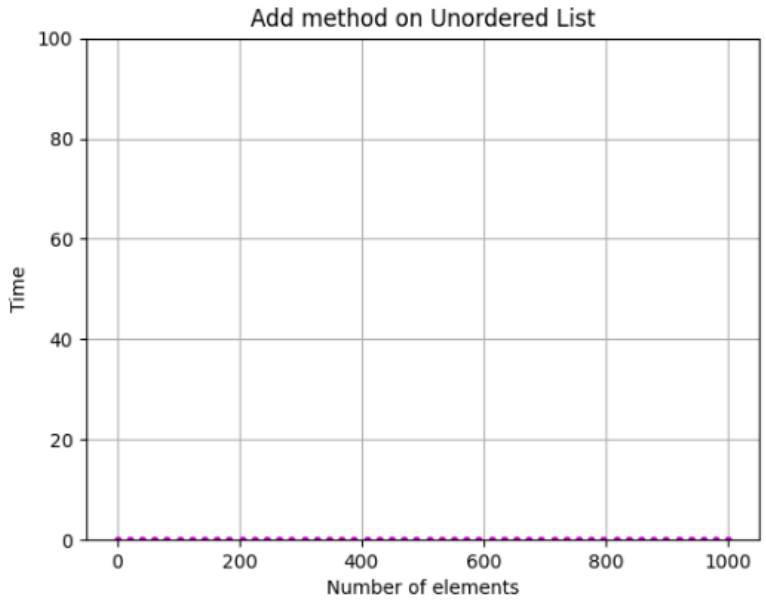








Sprawdziłyśmy też jak wygląda sytuacja z metodą add dla Unordered List (dla pythonowej nie mamy add).



Widzimy, że w większości przypadków pythonowa lista jest bardziej wydajna. Jedynie dla metody add Unordered List ma złożoność O(1) – nie zależy od liczby elementów. Dla reszty złożoność wynosi O(n).