Algorytmy i stuktury danych - Lista 4

Alicja Myśliwiec - gr. wtorek 7:30

In [1]:
```python
import time, random
import matplotlib.pyplot as plt
import numpy as np
```

Zad. 1 Zaimplementuj kolejkę przy użyciu pythonowych list w taki sposób, aby:

- koniec kolejki znajdował się na końcu listy,

In [2]:
```python
class QueueBaB(object):
    def __init__(self):
        self.list_of_items = []

    def isEmpty(self):
        if not self.list_of_items:
            return True
        return False

    def enqueue(self, *items):              #umożliwia to dodanie wielu elemntów na raz do kolejki
        for item in list(items):
            self.list_of_items.append(item)

    def dequeue(self):
        return self.list_of_items.pop(0)

    def size(self):
        return len(self.list_of_items)

    def __str__(self):
        return str(self.list_of_items)
```

In [3]:
```python
b = QueueBaB()
b.enqueue(3)
b.enqueue(4)
b.enqueue(5, 7, 9, 10)
b.dequeue()
print(b)
```

```
[4, 5, 7, 9, 10]
```

In [4]:
```python
b = QueueBaB()
b.enqueue(1)
print(b.isEmpty())
b.dequeue()
print(b.isEmpty())
```

```
False
True
```

- koniec kolejki znajdował się na początku listy.

In [5]:
```python
class QueueBaE(object):
    def __init__(self):
        self.list_of_items = []

    def isEmpty(self):
        if not self.list_of_items:
            return True
        return False

    def enqueue(self, *items):
        for item in list(items):
            self.list_of_items.insert(0, item)

    def dequeue(self):
        return self.list_of_items.pop()

    def size(self):
        return len(self.list_of_items)

    def __str__(self):
        return str(self.list_of_items)
```

In [6]:
```python
b = QueueBaE()
b.enqueue(3)
b.enqueue(4)
b.enqueue(5, 7, 9, 10)
b.dequeue()
print(b, b.size())
```

```
[10, 9, 7, 5, 4] 5
```

Zad. 2 Zaprojektuj i przeprowadź eksperyment porównujący wydajność obu implementacji.

W poniższym zadaniu porównam metody enqueue, dequeue oraz obie jednoczesnie dla obu kolejek

In [7]:
```python
def enqueue_times(n):

    BaB = QueueBaB()
    BaE = QueueBaE()

    start_BaB = time.time()
    for i in range(0, n):
        BaB.enqueue(i)
    end_BaB = time.time()
    time_BaB = end_BaB - start_BaB

    start_BaE = time.time()
    for i in range(0, n):
        BaE.enqueue(i)
    end_BaE = time.time()
    time_BaE = end_BaE- start_BaE

    return [time_BaB, time_BaE], [BaB, BaE]
```

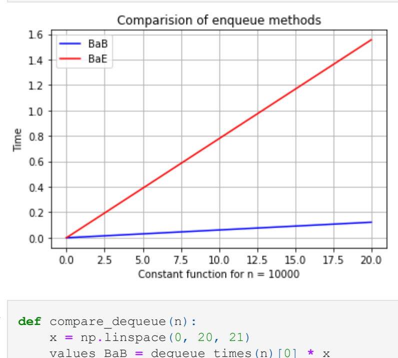
In [8]:
```python
def dequeue_times(n):

    BaB = enqueue_times(n)[1][0]
    BaE = enqueue_times(n)[1][1]

    start_BaB = time.time()
    for i in range(0, n):
        BaB.dequeue()
    end_BaB = time.time()
    time_BaB = end_BaB - start_BaB

    start_BaE = time.time()
    for i in range(0, n):
        BaE.dequeue()
    end_BaE = time.time()
    time_BaE = end_BaE - start_BaE

    return [time_BaB, time_BaE]
```

In [88]:
```python
def enq_and_deq(n):
    BaB = QueueBaB()
    BaE = QueueBaE()

    start_BaB = time.time()
    for i in range(0, n):
        BaB.enqueue(i)
    for i in range(0, n):
        BaB.dequeue()
    end_BaB = time.time()
    time_BaB = end_BaB - start_BaB

    start_BaE = time.time()
    for i in range(0, n):
        BaE.enqueue(i)
    for i in range(0, n):
        BaE.dequeue()
    end_BaE = time.time()
    time_BaE = end_BaE - start_BaE

    return [time_BaB, time_BaE]
```

Teraz, dla konkretnej liczby n wywolan, stworze funkcje stałą. Następnie sprawdzimy czy faktycznie zależność jest liniowa.

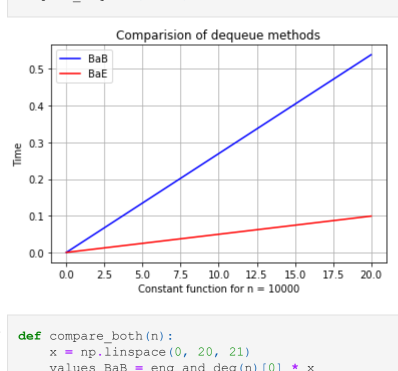
In [124...
```python
def compare_enqueue(n):
    x = np.linspace(0, 20, 21)
    values_BaB = enqueue_times(n)[0][0] * x
    values_BaE = enqueue_times(n)[0][1] * x
    plt.plot(x, values_BaB, c="blue", label="BaB")
    plt.plot(x, values_BaE, c="red", label="BaE")
    plt.title("Comparison of enqueue methods")
    plt.xlabel("Constant function for n = {}".format(n))
    plt.ylabel("Time")
    plt.legend()
    plt.grid()
    plt.show()
```

In [125...
```python
compare_enqueue(10000)
```



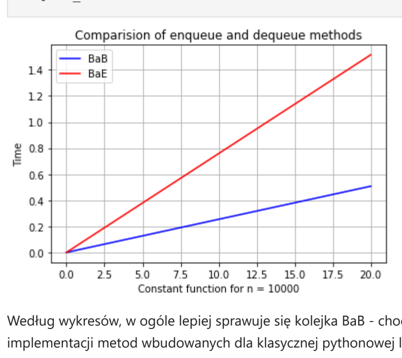
In [130...
```python
def compare_dequeue(n):
    x = np.linspace(0, 20, 21)
    values_BaB = dequeue_times(n)[0] * x
    values_BaE = dequeue_times(n)[1] * x
    plt.plot(x, values_BaB, c="blue", label="BaB")
    plt.plot(x, values_BaE, c="red", label="BaE")
    plt.title("Comparison of dequeue methods")
    plt.xlabel("Constant function for n = {}".format(n))
    plt.ylabel("Time")
    plt.legend()
    plt.grid()
    plt.show()
```

In [131...
```python
compare_dequeue(10000)
```



In [128...
```python
def compare_both(n):
    x = np.linspace(0, 20, 21)
    values_BaB = enq_and_deq(n)[0] * x
    values_BaE = enq_and_deq(n)[1] * x
    plt.plot(x, values_BaB, c="blue", label="BaB")
    plt.plot(x, values_BaE, c="red", label="BaE")
    plt.title("Comparison of enqueue and dequeue methods")
    plt.xlabel("Constant function for n = {}".format(n))
    plt.ylabel("Time")
    plt.legend()
    plt.grid()
    plt.show()
```

In [129...
```python
compare_both(10000)
```



Według wykresów, w ogóle lepiej sprawuje się kolejka BaB - choć przy metodzie dequeue jest wolniejsza. osobne wyniki metod wynikają z implementacji metod wbudowanych dla klasycznej pythonowej listy.

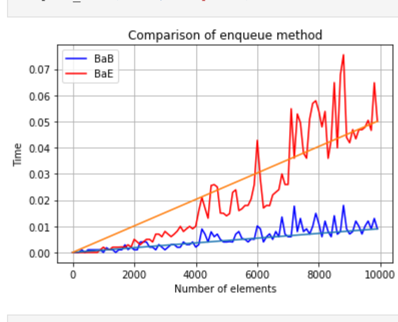
Teraz sprawdzę jak wygląda wykres sprawdzając czas dla każdego n po kolei

In [132...
```python
def get_enqueue_times(n):
    single_times_BaB, single_times_BaE = [], []
    x = [x for x in range(0, n, 100)]
    for timer in x:
        times = enqueue_times(timer)[0]
        BaB_times = times[0]
        BaE_times = times[1]
        single_times_BaB.append(BaB_times)
        single_times_BaE.append(BaE_times)

    return [x, single_times_BaB, single_times_BaE]
```

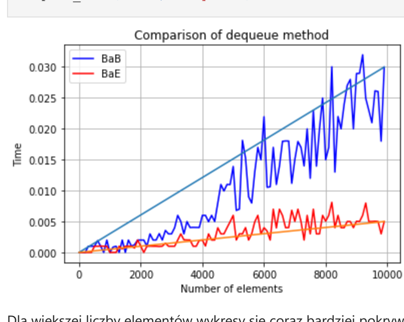
In [133...
```python
def get_dequeue_times(n):
    single_times_BaB, single_times_BaE = [], []
    x = [x for x in range(0, n, 100)]
    for timer in x:
        times = dequeue_times(timer)
        BaB_times = times[0]
        BaE_times = times[1]
        single_times_BaB.append(BaB_times)
        single_times_BaE.append(BaE_times)

    return [x, single_times_BaB, single_times_BaE]
```

In [134...
```python
def compare_time(n, name):
    if name == "enqueue":
        data = get_enqueue_times(n)
        tlt = "Comparison of enqueue method"
    elif name == "dequeue":
        data = get_dequeue_times(n)
        tlt = "Comparison of dequeue method"

    x = data[0]
    values_BaB = data[1]
    values_BaE = data[2]
    plt.plot(x, values_BaB, c="blue", label="BaB")
    plt.plot(x, values_BaE, c="red", label="BaE")

    xaxis = [0, x[-1]]                      # wykresy liniowe
    plt.plot(xaxis, [0, values_BaB[-1]])
    plt.plot(xaxis, [0, values_BaE[-1]])

    plt.title(tlt)
    plt.xlabel("Number of elements")
    plt.ylabel("Time")
    plt.legend()
    plt.grid()
    plt.show()
```

In [135...
```python
compare_time(10000, "enqueue")
```

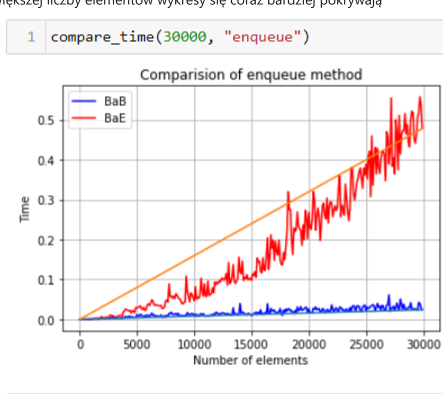


In [136...
```python
compare_time(10000, "dequeue")
```

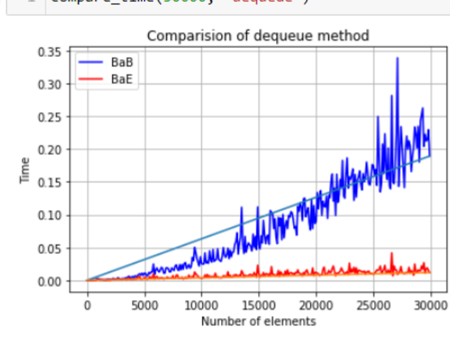


Dla większej liczby elementów wykresy się coraz bardziej pokrywają

```python
compare_time(30000, "enqueue")
```




```python
compare_time(30000, "dequeue")
```

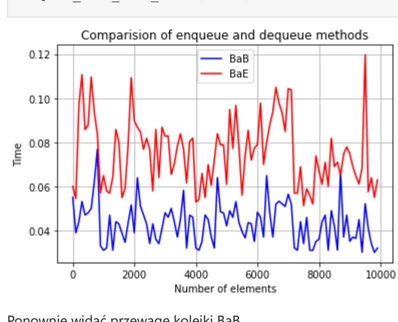


Można też zobrazować, w jakich przedziałach oscylują wykonania dla tej samej liczby elementów.

In [22]:
```python
def get_both_times(n):
    single_times_BaB, single_times_BaE = [], []
    x = [x for x in range(0, n, 100)]
    for timer in x:
        times = enq_and_deq(n)
        BaB_times = times[0]
        BaE_times = times[1]
        single_times_BaB.append(BaB_times)
        single_times_BaE.append(BaE_times)

    return [x, single_times_BaB, single_times_BaE]
```

In [23]:
```python
def compare_both_same_times(n):
    data = get_both_times(n)
    x = data[0]
    values_BaB = data[1]
    values_BaE = data[2]
    plt.plot(x, values_BaB, c="blue", label="BaB")
    plt.plot(x, values_BaE, c="red", label="BaE")
    plt.title("Comparison of enqueue and dequeue methods")
    plt.xlabel("Number of executions")
    plt.ylabel("Time")
    plt.legend()
    plt.grid()
    plt.show()
```

In [24]:
```python
compare_both_same_times(10000)
```



Ponownie widać przewagę kolejki BaB

Zad. 3 Rozważ sytuację z życia wziętą, np.:

- auta w kolejce do myjni,
- kasy w supermarkecie,
- samoloty na pasie startowym,
- okienko w banku.

Postaw pytanie badawcze. Wykorzystując liniowe struktury danych zaprojektuj i przeprowadź symulację, która udzieli na nie odpowiedzi. Pamiętaj o określeniu wszystkich uproszczeń swojego modelu.

Bistro

Z uwagi na panujące obostrzenia, w pewnym bistro jest określona maksymalna liczba osób, która może przebywać jednocześnie w lokalu. Symulacje przeprowadzane są przy założeniu, iż restrykcje się zmieniają oraz pozostaje dysproporcja w tempie pracy poszczególnych kasjerek. Czas obsługi konkretnego klienta również jest zróżnicowany (np. nakłada większą ilość jedzenia). Z zeszłorocznych danych również wiemy, że bistro odwiedza od 8 do 15 osób na godzinę.

Każdy klient, który w godzinę zamknięcia bistro stoi w kolejce poza lokalem, niestety nie może zostać obsłużony. Ile średnio osób dziennie, nie zostaje obsługiwanych?

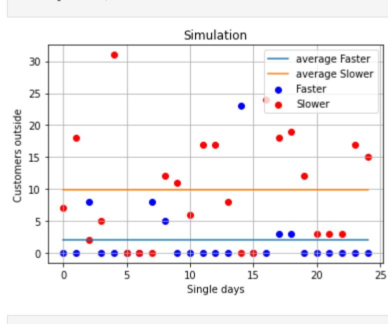
In [137...
```python
class Bistro:
    def __init__(self, restriction, faster_service_pace):
        """
        :param restriction: how many clients can wait inside, pandemic times :(
        :param faster_service_pace: client can be serviced by different cashiers
        True - faster, False - slower cashier!
        """
        self.clients = None
        self.current_client = None
        self.open_hours = 360
        self.full_queue_inside = False
        self.restricted_amount_of_clients = restriction
        self.faster_service_pace = faster_service_pace

    def new_clients(self):
        """
        Function determine how many clients will arrive in specific hour
        :return: True if it's a full hour, if not - False
        """
        if self.open_hours in [60, 120, 180, 240, 300, 360]:
            number = random.randint(8, 15)
            self.clients = number
            return True
        return False

    def every_minute_tick(self):
        self.open_hours -= 1

    def next_to_the_counter(self, new_client):
        """
        next customer is served
        """
        self.current_client = new_client
```

In [138...
```python
class Client:
    def __init__(self, spent_time):
        """
        :param spent_time: Time that single client will be served
        """
        self.time_remain = spent_time

    def waiting(self):
        self.time_remain -= 1

    def has_been_served(self):
        """
        Function determine whether client can leave the queue
        :return:
        """
        if self.time_remain == 0:
            return True
        return False
```

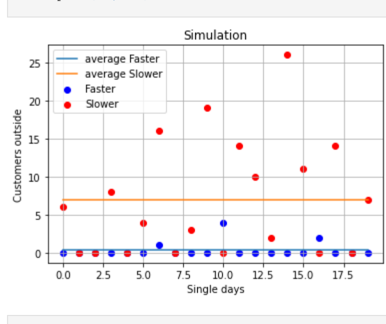
In [139...
```python
def single_simulation(restrict, fsp):
    """
    :return: amount of people outside
    """
    bistro = Bistro(restrict, fsp)
    queue = QueueBaE()

    next_client = None
    clients_outside = 0
    client_time_range = [4, 5, 6, 7]
    client_time = random.choice(client_time_range)

    if not bistro.faster_service_pace:
        client_time += random.randint(1, 2)  # slower cashier makes service time longer

    while bistro.open_hours > 0:

        if bistro.new_clients():

            for it in range(bistro.clients):
                client = Client(client_time)
                queue.enqueue(client)

        if queue.isEmpty():  # nobody in queue, time keep going
            bistro.every_minute_tick()
            continue

        if bistro.current_client is None or next_client.has_been_served():
            next_client = queue.dequeue()
            bistro.next_to_the_counter(next_client)

        if queue.size() >= bistro.restricted_amount_of_clients:
            bistro.full_queue_inside = True
        else:
            bistro.full_queue_inside = False

        next_client.waiting()
        bistro.every_minute_tick()

    if bistro.full_queue_inside:
        clients_outside = queue.size() - restrict

    return clients_outside
```

In [149...
```python
def interpret(restrict, n):
    """
    In this function I am going to interpret _1 as calculations for faster cashier, similary _2 for the slower
    :param n: how many time to simulate a single day
    """
    list_of_clients_outside_1, list_of_clients_outside_2, x = [], [], []
    average_amount_1, average_amount_2 = 0, 0

    for i in range(n):
        data_1 = single_simulation(restrict, True)
        list_of_clients_outside_1.append(data_1)
        data_2 = single_simulation(restrict, False)
        list_of_clients_outside_2.append(data_2)

        x.append(i)

    for num in list_of_clients_outside_1:
        average_amount_1 += num

    for num in list_of_clients_outside_2:
        average_amount_2 += num

    result_1, result_2 =  average_amount_1 / len(list_of_clients_outside_1), average_amount_2 / len(list_of_cli

    values_1 = list_of_clients_outside_1
    values_2 = list_of_clients_outside_2
    plt.scatter(x, values_1, c="blue", label="Faster")
    plt.scatter(x, values_2, c="red", label="Slower")

    plt.plot(x, [result_1 for i in range(n)], label="average Faster")
    plt.plot(x, [result_2 for i in range(n)], label="average Slower")

    plt.title("Simulation")
    plt.xlabel("Single days")
    plt.ylabel("Customers outside")
    plt.legend()
    plt.grid()
    plt.show()
```

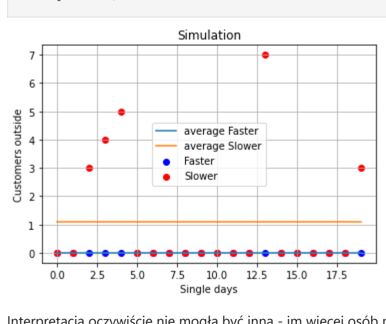
In [150...
```python
interpret(6, 25)
```



In [151...
```python
interpret(10, 20)
```



In [152...
```python
interpret(23, 20)
```



Interpretacja oczywiście nie mogła być inna - im więcej osób może przebywać w lokalu, tym więcej ich obsłużymy po zamknięciu (średnia ilość osób w kolejce poza bistro). Również tempo kasjerki sprawia, że w ciągu dnia jest mniej lub więcej 'nieobsłużonych' klientów.

Zad. 4 Napisz program, który sprawdzi poprawność składni dokumentu HTML pod kątem brakujących znaczników zamykających.

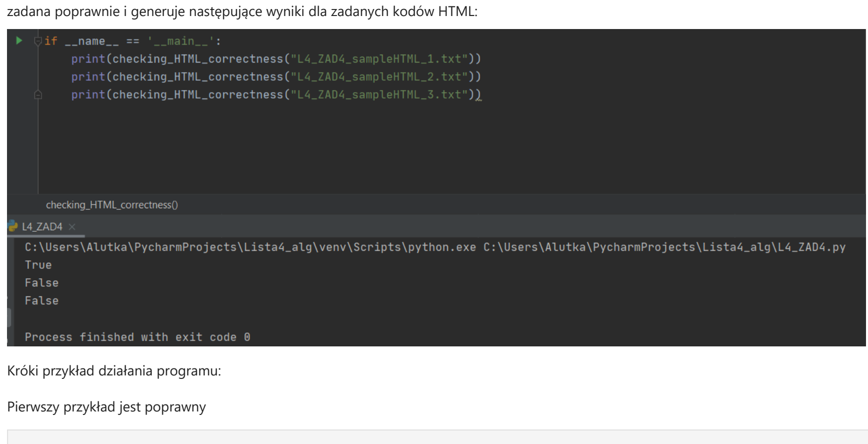
Program zacznę od zaimplementowania stosu

In [34]:
```python
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items) - 1]

    def size(self):
        return len(self.items)

    def __getitem__(self, item):
        return self.items[item]

    def __str__(self):
        return str(self.items)
```

Następnie, funkcja właściwa. Wydziela ona elementy zanjdujące się pomiędzy znacznikami "<" oraz ">". Jeśli jest to komentarz - usuwa go w całości (w komenarzu może być wszystko, tzn. przykładowo pootwierane znaczniki). Odpowiednio wyselekcjonowane elementy funkcja kolejno lub wrzuca na stos, lub je ściąga.

In [35]:
```python
def checking_HTML_correctness(text):

    interesting_list = []
    checking_stack = Stack()

    while '>' in text:

        start = text.find("<") + len("<")
        end = text.find(">")
        end_comment = text.find("-->")
        substring = text[start:end]

        if substring[0:3] == "!--":
            text = text[0:start - 1] + text[end_comment + len("-->"):]
            continue

        if len(substring.split("<")) == 1:
            interesting_list.append(substring)
            text = text[0:start - 1] + text[end + 1:]

        else:
            start2 = substring.find("<") + len("<")
            subsubstring = substring[start2:end]
            interesting_list.append(subsubstring)
            text = text[0:start + len(substring.split("<")[0])] + text[end + 1:]

    for element in interesting_list:
        helping_list = element.split(" ")
        ele = helping_list[0]
        if ele in ['meta', 'link', 'img', '!DOCTYPE', 'br', 'hr', 'col', 'command', 'input', 'base', 'area']:
            continue
        if not checking_stack.isEmpty():
            if '/' + checking_stack[-1] == ele:
                checking_stack.pop()
            else:
                checking_stack.push(ele)
        else:
            checking_stack.push(ele)

    if checking_stack.isEmpty():
        return True
    return False
```

Funkcja ta dla potrzeb działania w Jupyterze została zmodyfikowana tak, aby przyjmowała string nie plik jako argument. W pliku .py jest zadana poprawnie i generuje następujące wyniki dla zadanych kodów HTML:



```
if __name__ == '__main__':
    print(checking_HTML_correctness("L4_ZAD4_sampleHTML_1.txt"))
    print(checking_HTML_correctness("L4_ZAD4_sampleHTML_2.txt"))
    print(checking_HTML_correctness("L4_ZAD4_sampleHTML_3.txt"))

checking_HTML_correctness()
L4_ZAD4
C:\Users\Alutka\PycharmProjects\Lista4_alg\venv\Scripts\python.exe C:\Users\Alutka\PycharmProjects\Lista4_alg\L4_ZAD4.py
True
False
False

Process finished with exit code 0
```

Króki przykład działania programu:

Pierwszy przykład jest poprawny

In [36]:
```python
sample_1 = "<head> <p> </p> <meta> <!-- --> </head>"
checking_HTML_correctness(sample_1)
```

Out[36]: True

W drugim natomiast brakuje jednego domknięcia "\</div>". Dlatego otrzymujemy False

In [37]:
```python
sample_2 = "<footer> <div> <div> <div> <ul> <!-- Email ====== --><!-- Facebook ======== --></ul> <p>Copyright <
checking_HTML_correctness(sample_2)
```

Out[37]: False

Dodajmy do przykładu brakujący zamykający "\</div>". Teraz wszytsko jest w porządku.

In [38]:
```python
sample_2_repaired = "<footer> <div> <div> <div> <ul> <!-- Email ====== --> <!-- Facebook ======== --> </ul> <p>
checking_HTML_correctness(sample_2_repaired)
```

Out[38]: True

Ostatni przykład, w którym widać że zawartość komentarzy nie ma znaczenia.

In [39]:
```python
sample_3 = "<head> <a> </a> <!-- <title>  </p> --> </head>"
checking_HTML_correctness(sample_3)
```

Out[39]: True

Zad. 5 Dodaj brakujące metody do klasy `UnorderedList` prezentowanej na wykładzie.

In [40]:
```python
class Node:
    def __init__(self, init_data):
        self.data = init_data
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, new_data):
        self.data = new_data

    def setNext(self, new_next):
        self.next = new_next
```

```
Time for unrordered list: 1.0464789867401123

Time for python list: 0.0019931793212890625
Time for unrordered list: 0.9790396690368652

Time for python list: 0.0032656192779541016
Time for unrordered list: 0.8619673252105713
```

In [57]:
```python
def pop_times(n):
    python_list = [0 for i in range(0, n)]
    un_list = UnorderedList()
    for i in range(0, n):
        un_list.add(i)

    start_python = time.time()
    for i in range(n):
        python_list.pop()
    end_python = time.time()
    time_python = end_python - start_python

    start_un = time.time()
    for i in range(n):
        un_list.pop()
    end_un = time.time()
    time_un = end_un - start_un

    return [time_python, time_un]
```

In [58]:
```python
for i in range(5):
    check = pop_times(2000)
    print("Time for python list: {}".format(check[0])+"\n"+"Time for unrordered list: {}". format(check[1])+"\n")
```

```
Time for python list: 0.0009975433349609375
Time for unrordered list: 2.442253828048706

Time for python list: 0.0
Time for unrordered list: 2.7217960357666016

Time for python list: 0.0
Time for unrordered list: 2.795039653778076

Time for python list: 0.0
Time for unrordered list: 2.5997231006622314

Time for python list: 0.000997304916381836
Time for unrordered list: 2.5144569873809814
```

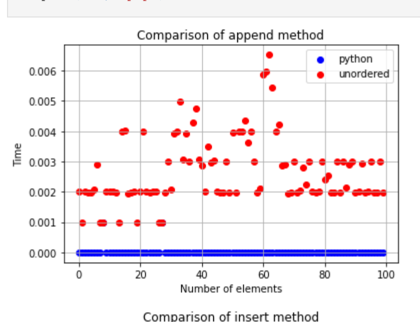
Z powyższych symulacji dla aż 2000 wywołań, widać że lista wbudowana jest dużo wydajniejsza. Co do listy jednokierunkowej najlepiej jednak spisuje się metoda insert - czas schodzi poniżej sekundy. Najsłabiej za to wypada metoda pop, której wywołanie dla n=2000 dochodzi do 3s.

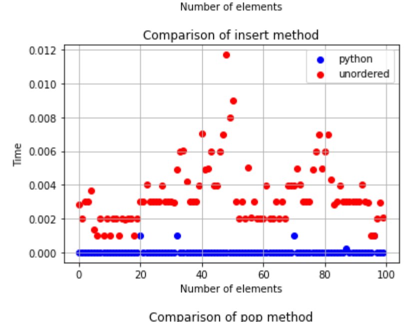
Wizualizacja:

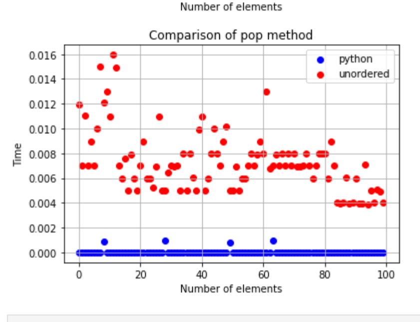
In [75]:
```python
def compare(n, name):

    x = [i for i in range(n)]
    values_python, values_un = [], []

    for timer in range(n):

        if name == 'pop':
            data = pop_times(n)
            ttl = 'Comparison of pop method'
        elif name == 'insert':
            data = insert_times(n)
            ttl = 'Comparison of insert method'
        elif name == 'append':
            data = append_times(n)
            ttl = 'Comparison of append method'
        elif name == 'experiment':          # na później :)
            data = experiment(n)
            ttl = 'Comparison of the three methods'

        values_python.append(data[0])
        values_un.append(data[1])
    plt.scatter(x, values_python, c="blue", label="python")
    plt.scatter(x, values_un, c="red", label="unordered")
    plt.title(ttl)
    plt.xlabel("Number of elements")
    plt.ylabel("Time")
    plt.legend()
    plt.grid()
    plt.show()
```

Ponieważ na wykonanie wykresu dla n=2000 czas oczekiwania jest dość duży i generuje duże obciążenie, wykres wykonamy dla mniejszych wartości n:

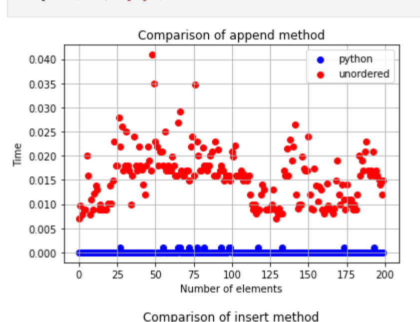
In [76]:
```python
compare(100, 'append')
compare(100, 'insert')
compare(100, 'pop')
```

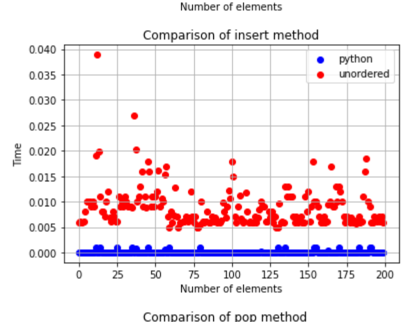


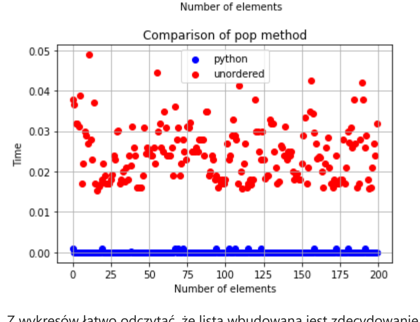




In [77]:
```python
compare(200, 'append')
compare(200, 'insert')
compare(200, 'pop')
```





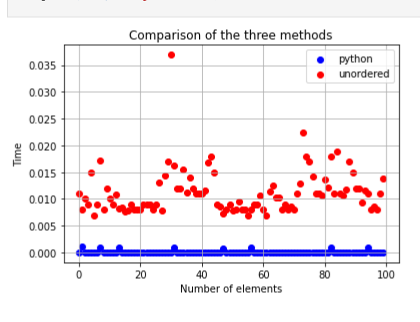


Z wykresów łatwo odczytać, że lista wbudowana jest zdecydowanie szybsza.

Przykład wykonania wszystkich operacji

In [78]:
```python
def experiment(n):
    python_list = []
    un_list = UnorderedList()

    start_python = time.time()
    for i in range(0, n):
        python_list.append(n)
        python_list.insert(0, i)
        python_list.pop()
    end_python = time.time()
    time_python = end_python - start_python

    start_un = time.time()
    for i in range(0, n):
        un_list.append(i)
        un_list.insert(0, i)
        un_list.pop()
    end_un = time.time()
    time_un = end_un - start_un

    return [time_python, time_un]
```

In [79]:
```python
compare(100, 'experiment')
```



LINK GITHUB

https://github.com/AlutkaMalutka/Programowanie_python/tree/main/semestr_3/Lista_4-3s-