Task 1

Egor Turukhanov

Group C4134

23.09.19

VECTORS GENERATION

At this stage we're generating 10 n-dimensional vector with sizes 50 to 500 with step = 50. I've used random integers until 1000 as elements of each vector. You can see my code of this operation below.

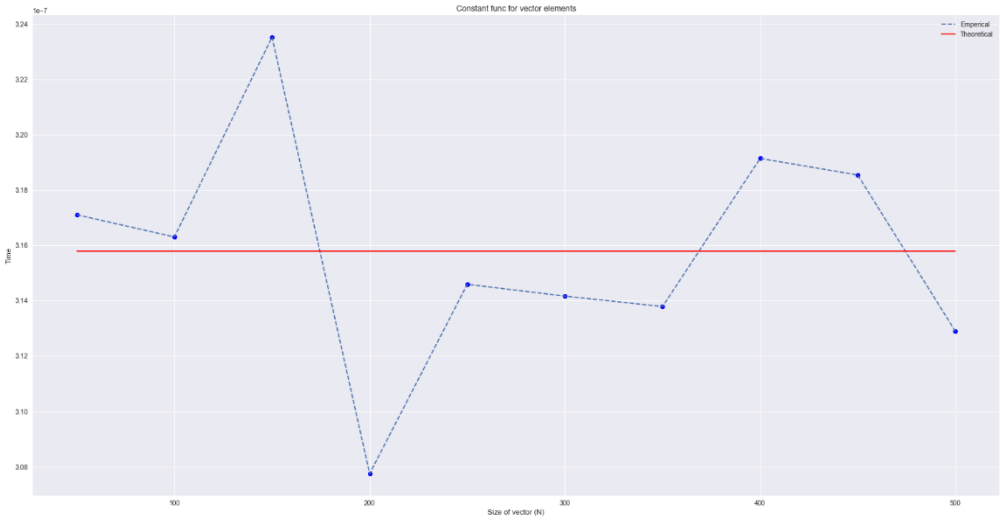
```
21 vectors = []
22 # Vectors Generation ####################
23 for i in range(50,  550, 50):
24     vector = np.random.randint(1000, size=i)
25     vectors.append(vector)
```

CONSTANT FUNCTION

Now we're applying constant function to our n-dimensional vector, measuring time 5 times for each vector and plotting graph of time complexity vs number of elements in vector comparing theoretical and empirical complexity of algorithm. For constant function complexity should be constant. Below you can see code and graph plotting.

```
28 # Constant ####################
29 def contant(data):
30     return(500)
31
32 total_time = []
33 vector_mean = pd.DataFrame(columns=['Time'])
34 for vector_i in vectors:
35     for i in range(1, 5, 1):
36         t = timeit.default_timer()
37         contant(vector_i)
38         elapsed_time = timeit.default_timer() - t
39         total_time.append(elapsed_time)
40     vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)
41
42 vector_mean.index = range(50,  550, 50)
43 fig = plt.figure()
44 ax1 = fig.add_subplot(111)
45 ax1.set_ylabel('Time')
46 ax1.set_xlabel('Size of vector (N)')
47 ax1.set_title('Constant func for vector elements')
48 plt.plot(vector_mean.index, vector_mean.Time, 'ob')
49 plt.plot(vector_mean.index, vector_mean.Time, '--', label='Emperical')
50 plt.plot(vector_mean.index, np.repeat(np.mean(vector_mean.Time), 10), 'r', label='Theoretical')
51 plt.legend()
52 plt.show()
```

Plot of results

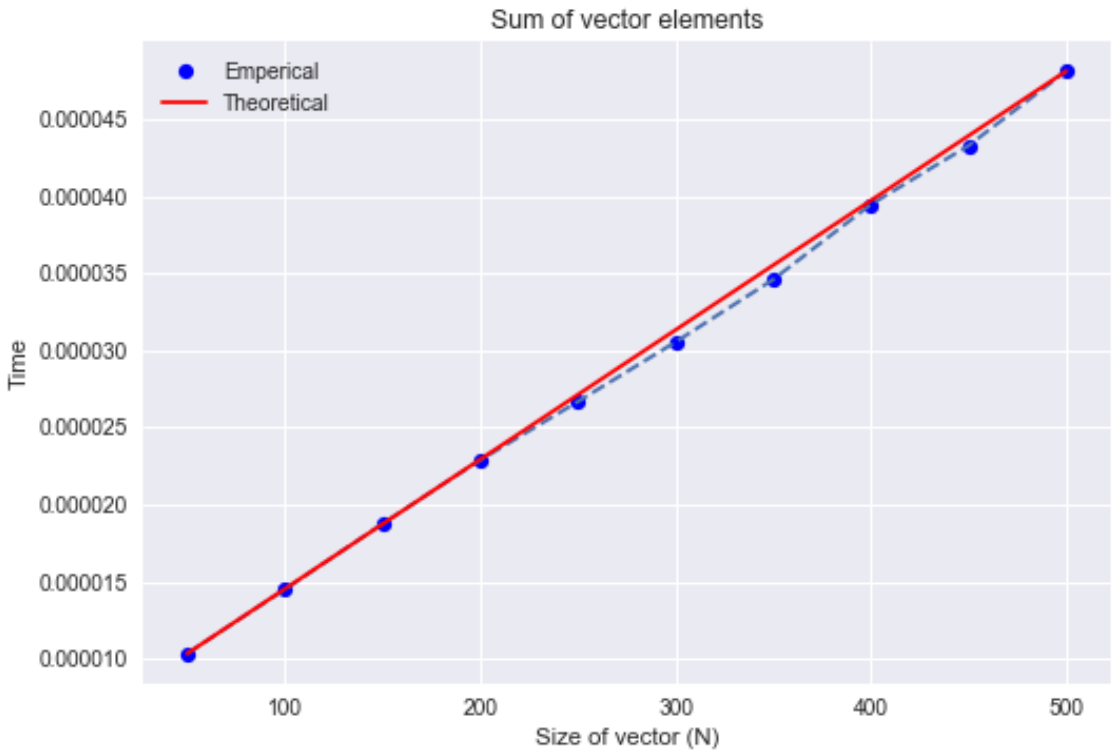


As we can see theoretical and empirical result show linear complexity of algorithm.

SUM OF ELEMENTS

Now we're computing sum of element in each vector. Its complexity should be linear (O(n)). Below you can see code and graph plotting.

```
# Sum ######################
total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for vector_i in vectors:
    for i in range(1, 5, 1):
        t = timeit.default_timer()
        sum(vector_i)
        elapsed_time = timeit.default_timer() - t
        total_time.append(elapsed_time)
    vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)

vector_mean.index = range(50,  550, 50)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Time')
ax1.set_xlabel('Size of vector (N)')
ax1.set_title('Sum of vector elements')
plt.plot(vector_mean.index, vector_mean.Time, 'ob', label='Emperical')
plt.plot(vector_mean.index, vector_mean.Time, '--')
plt.plot([min(vector_mean.index),max(vector_mean.index)], [min(vector_mean.Time),
          max(vector_mean.Time)], 'r', label='Theoretical')
plt.legend()
plt.show()
```

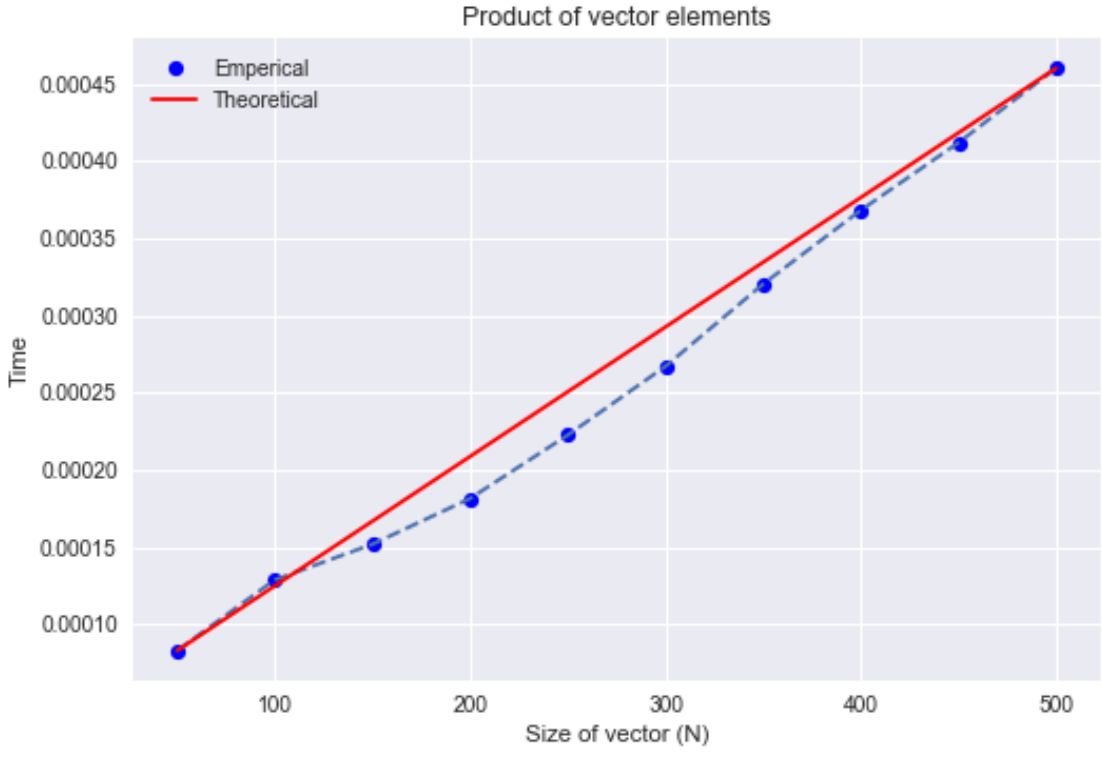


As we can see theoretical and empirical result show linear complexity (O(n)) of algorithm.

PRODUCT OF ELEMENTS

Now we're product of elements in each vector. Complexity of algorithm also should be linear (O(n)) as sum of elements. Below you can see code and graph plotting.

```
# Geom ######################

def geom(data):
    geom=1
    for i in data:
        geom=data*i
    return(geom)

total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for vector_i in vectors:
    for i in range(1, 5, 1):
        t = timeit.default_timer()
        geom(vector_i)
        elapsed_time = timeit.default_timer() - t
        total_time.append(elapsed_time)
    vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)


vector_mean.index = range(50,  550, 50)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Time')
ax1.set_xlabel('Size of vector (N)')
ax1.set_title('Product of vector elements')
plt.plot(vector_mean.index, vector_mean.Time, 'ob', label='Emperical')
plt.plot(vector_mean.index, vector_mean.Time, '--')
plt.plot([min(vector_mean.index),max(vector_mean.index)],
         [min(vector_mean.Time),max(vector_mean.Time)], 'r', label='Theoretical')
plt.legend()
plt.show()
```

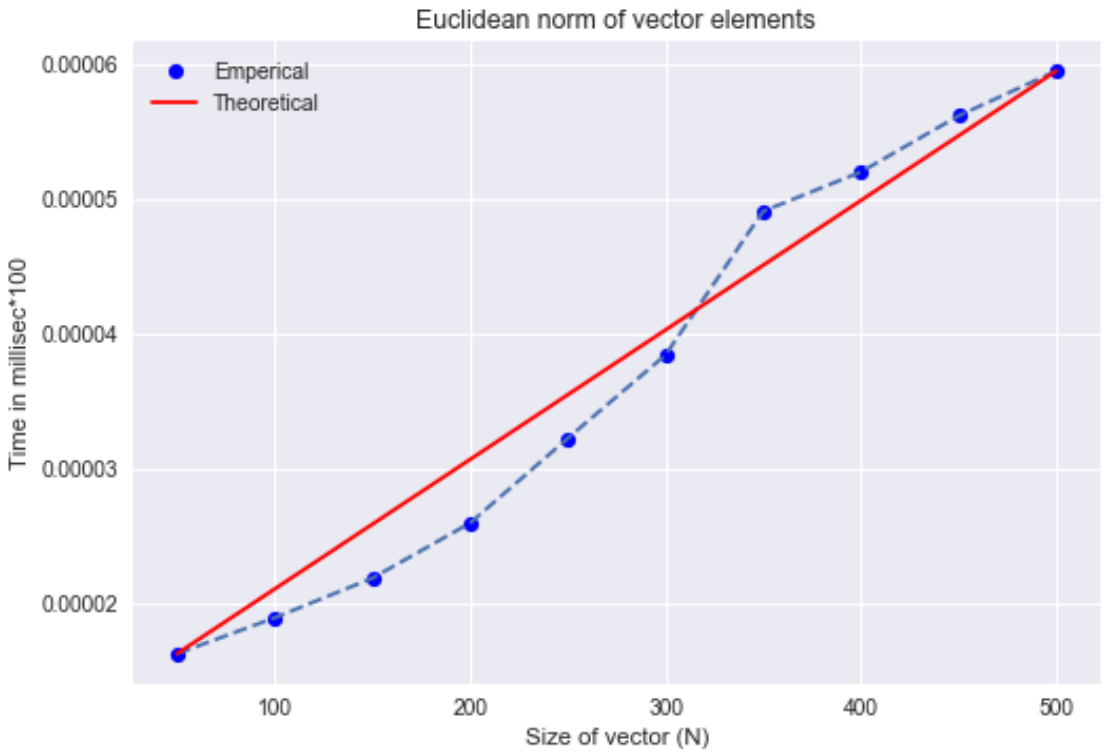


As we can see theoretical and empirical result show linear complexity (O(n)) of algorithm.

EUCLIDIAN NORM

Now we're computing Euclidean norm of elements in each vector. It should be linear (O(n)) as sum and product of elements. Below you can see code and graph plotting.

```
# Euclidian norm ######################

total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for vector_i in vectors:
    for i in range(1, 5, 1):
        t = timeit.default_timer()
        np.sqrt(sum(vector_i**2))
        elapsed_time = timeit.default_timer() - t
        total_time.append(elapsed_time)
    vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)

vector_mean.index = range(50,  550, 50)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Time in millisec*100')
ax1.set_xlabel('Size of vector (N)')
ax1.set_title('Euclidean norm of vector elements')
plt.plot(vector_mean.index, vector_mean.Time, 'ob', label='Emperical')
plt.plot(vector_mean.index, vector_mean.Time, '--')
plt.plot([min(vector_mean.index),max(vector_mean.index)],
         [min(vector_mean.Time),max(vector_mean.Time)], 'r', label='Theoretical')
plt.legend()
plt.show()
```

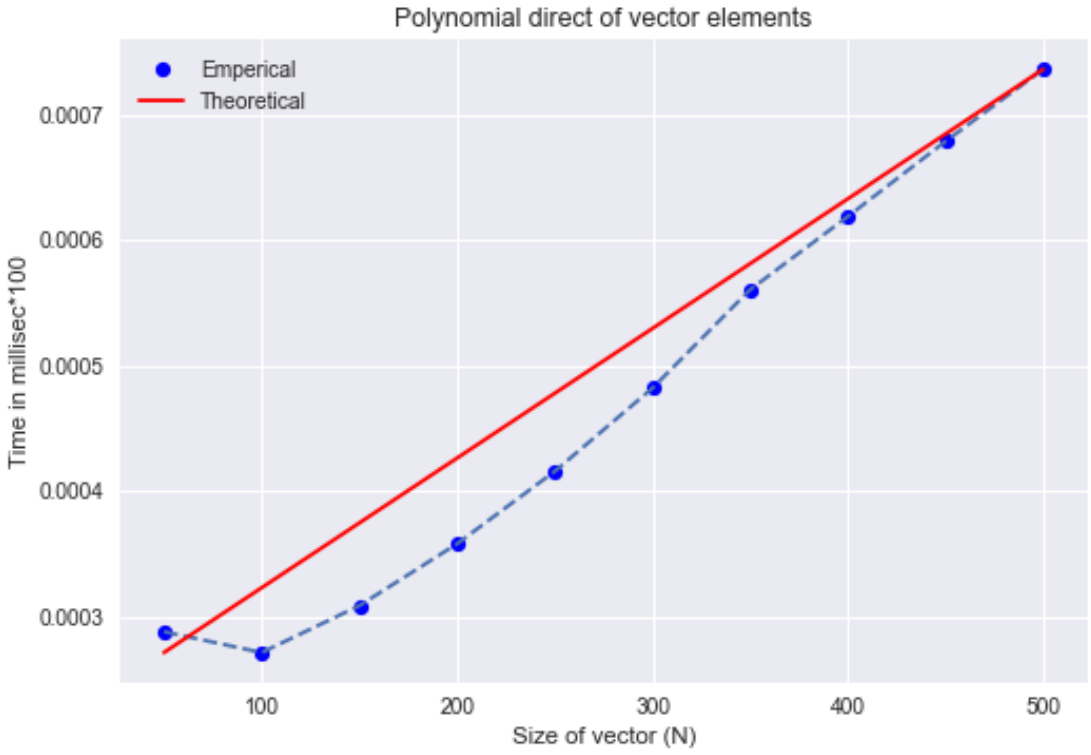


As we can see theoretical and empirical result show linear complexity (O(n)) of algorithm.

POLYMIAL DIRECT

Now we're computing direct polynomial of element in each vector. For direct polynomial function complexity should be linear (O(n)) as sum and product of elements. Below you can see code and graph plotting.

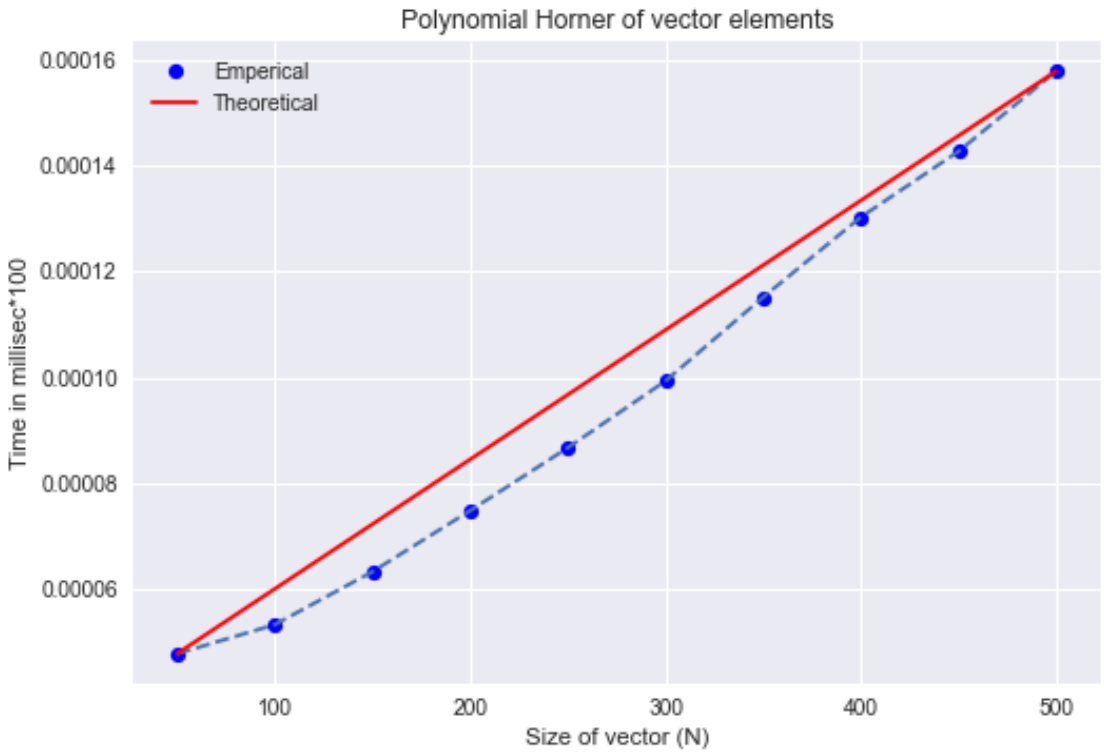
```python
# Polynom ######################
def poly_naive(A, x):
    p = 0
    for i, a in enumerate(A):
        p += (x ** i) * a
    return p


total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for vector_i in vectors:
    for i in range(1, 5, 1):
        pol_sum = 0
        t = timeit.default_timer()
        poly_naive(vector_i, 1.5)
        elapsed_time = timeit.default_timer() - t
        total_time.append(elapsed_time)
    vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)

vector_mean.index = range(50,  550, 50)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Time in millisec*100')
ax1.set_xlabel('Size of vector (N)')
ax1.set_title('Polynomial direct of vector elements')
plt.plot(vector_mean.index, vector_mean.Time, 'ob', label='Emperical')
plt.plot(vector_mean.index, vector_mean.Time, '--')
plt.plot([min(vector_mean.index),max(vector_mean.index)],
         [min(vector_mean.Time),max(vector_mean.Time)], 'r', label='Theoretical')
plt.legend()
plt.show()
```



Polynomial function computing tends to be linear complexity for empirical as well as for theoretical.

HORNER METHOD OF POLYMON CALCULATION

```
def poly_horner(A, x):
    p = A[-1]
    i = len(A) - 2
    while i >= 0:
        p = p * x + A[i]
        i -= 1
    return p


total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for vector_i in vectors:
    for i in range(1, 5, 1):
        pol_sum = 0
        t = timeit.default_timer()
        poly_horner(vector_i, 1.5)
        elapsed_time = timeit.default_timer() - t
        total_time.append(elapsed_time)
    vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)

vector_mean.index = range(50,  550, 50)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Time in millisec*100')
ax1.set_xlabel('Size of vector (N)')
ax1.set_title('Polynomial Horner of vector elements')
plt.plot(vector_mean.index, vector_mean.Time, 'ob', label='Emperical')
plt.plot(vector_mean.index, vector_mean.Time, '--')
plt.plot([min(vector_mean.index),max(vector_mean.index)],
         [min(vector_mean.Time),max(vector_mean.Time)], 'r', label='Theoretical')
plt.legend()
plt.show()
```

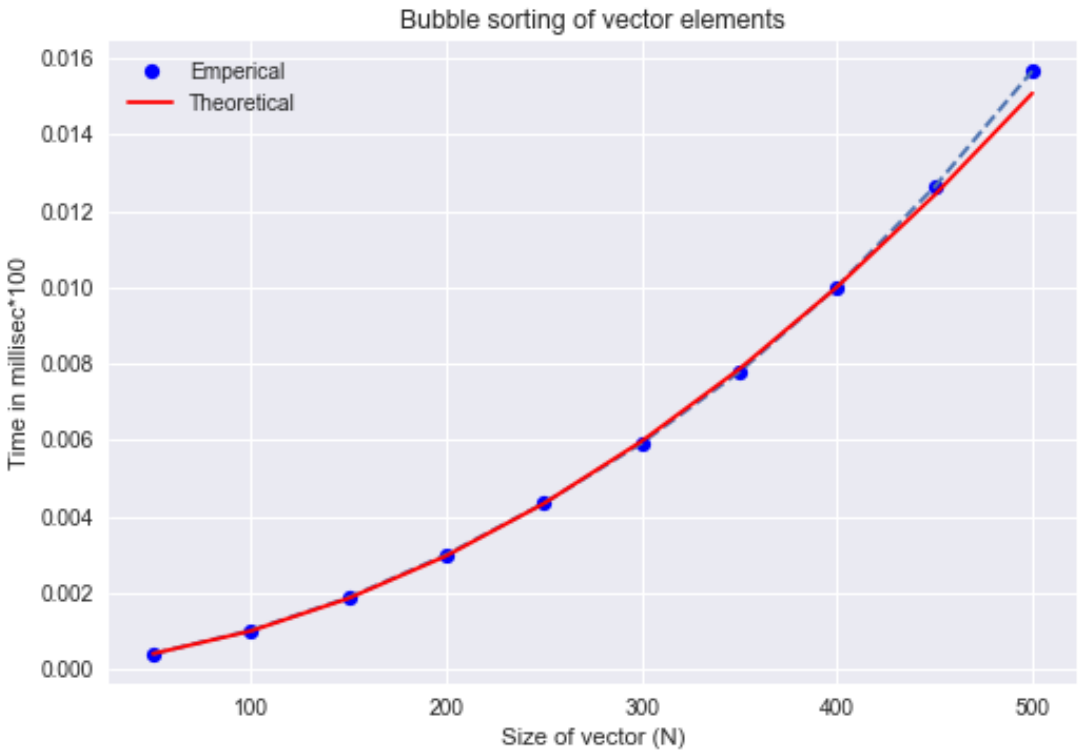


As a direct calculation of polynomial function Horner representation of polynomial function also has linear complexity.

BUBBLE SORT

Theoretically bubble sort algorithm should be quadratic complexity algorithm (O(n^2)), because it has 2 iteration though each element (loop FOR) of vector.

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]


total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for vector_i in vectors:
    for i in range(1, 5, 1):
        pol_sum = 0
        t = timeit.default_timer()
        bubbleSort(vector_i)
        elapsed_time = timeit.default_timer() - t
        total_time.append(elapsed_time)
    vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)

vector_mean.index = range(50,  550, 50)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Time in millisec*100')
ax1.set_xlabel('Size of vector (N)')
ax1.set_title('Bubble sorting of vector elements')
plt.plot(vector_mean.index, vector_mean.Time, 'ob', label='Emperical')
plt.plot(vector_mean.index, vector_mean.Time, '--')
plt.plot(vector_mean.index, np.arange(min(np.sqrt(vector_mean.Time)),
                                      max(np.sqrt(vector_mean.Time)),
          (max(np.sqrt(vector_mean.Time)))/11)**2, 'r', label='Theoretical')
plt.legend()
plt.show()
```



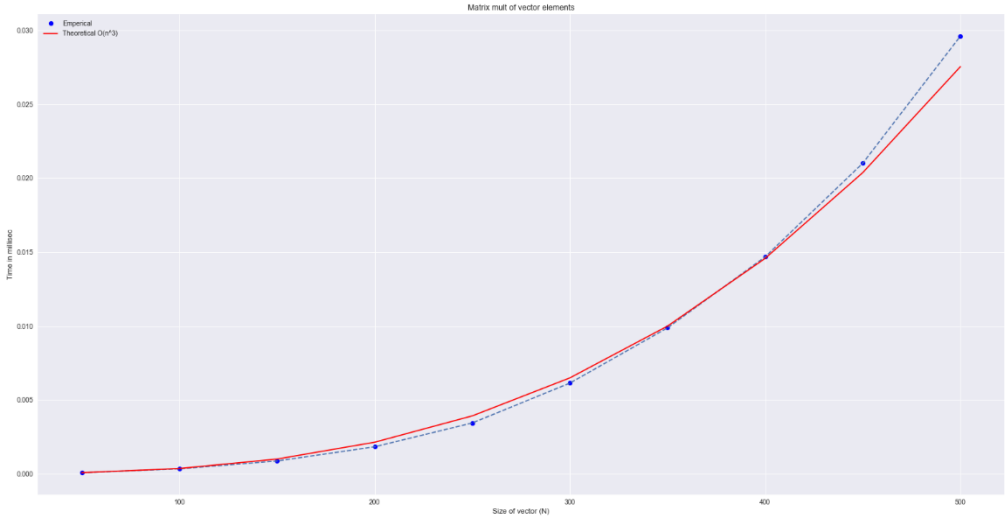
As can be seen from the plot above empirical complexity of bubble sort of algorithms tends to be quadratic as well as theoretical complexity.

MATRIX MULTIPLICATION

Matrix multiplication should be O(n^3), because it has 3 FOR loop for each element of vectors. The results can be seen below.

```
# Matrix Generation ####################
for i in range(50,  550, 50):

    matrix_A = np.random.randint(1000, size=(i, i))
    matrix_B = np.random.randint(1000, size=(i, i))
    matrixes.append([matrix_A, matrix_B])
```

```
total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for vector_i in matrixes:
    for i in range(1, 5, 1):
        pol_sum = 0
        t = timeit.default_timer()
        vector_i[0].dot(vector_i[1])
        elapsed_time = timeit.default_timer() - t
        total_time.append(elapsed_time)
    vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)

vector_mean.index = range(50,  550, 50)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Time in millisec')
ax1.set_xlabel('Size of vector (N)')
ax1.set_title('Matrix mult of vector elements')
plt.plot(vector_mean.index, vector_mean.Time, 'ob', label='Emperical')
plt.plot(vector_mean.index, vector_mean.Time, '--')
plt.plot(vector_mean.index, np.arange(min(np.cbrt(vector_mean.Time)),
                                      max(np.cbrt(vector_mean.Time)),
          (max(np.sqrt(vector_mean.Time))-min(np.cbrt(vector_mean.Time)))/4.5)**3,
    'r', label='Theoretical O(n^3)')
plt.legend()
plt.show()
```



Theoretical and empirical results of algorithm complexity for this case is also the same, as can be seen from the plot above.

CONCLUSION

Theoretical and empirical time complexity for most of our algorithms tend to be equal. Of course, for each computer and OS this time will differ. In our case, the most significant difference is between theoretical and empirical results is constant function. I think that difference is occur due to many reasons from background processes in my OS to temperature of hardware in my laptop.