11 Exercises

September 13, 2020

Import the modules that will be used.

[0]:
```python
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline
```

0.1 Exercise 11.1

Determine by counting the number of mathematical operations the complexity of:

1. Dot product between two vectors
2. Matrix-vector product
3. Matrix-matrix product

for vectors of length n and matrices of size $n \times n$.

This is a reasoning exercise - you do not need to write a program. Express your answers in text and using LaTeX in a Markdown cell.

0.1.1 Optional

Test the complexity experimentally with your own functions for performing the operations, and with the NumPy 'vectorised' equivalents.

YOUR ANSWER HERE

1. $O(1)$
2. $O(n)$
3. $O(n^2)$

0.2 Exercise 11.2

For the recursive factorial algorithm in Activity 04, determine the algorithmic complexity by inspecting your implementation of the algorithm. Test this against numerical experiments.

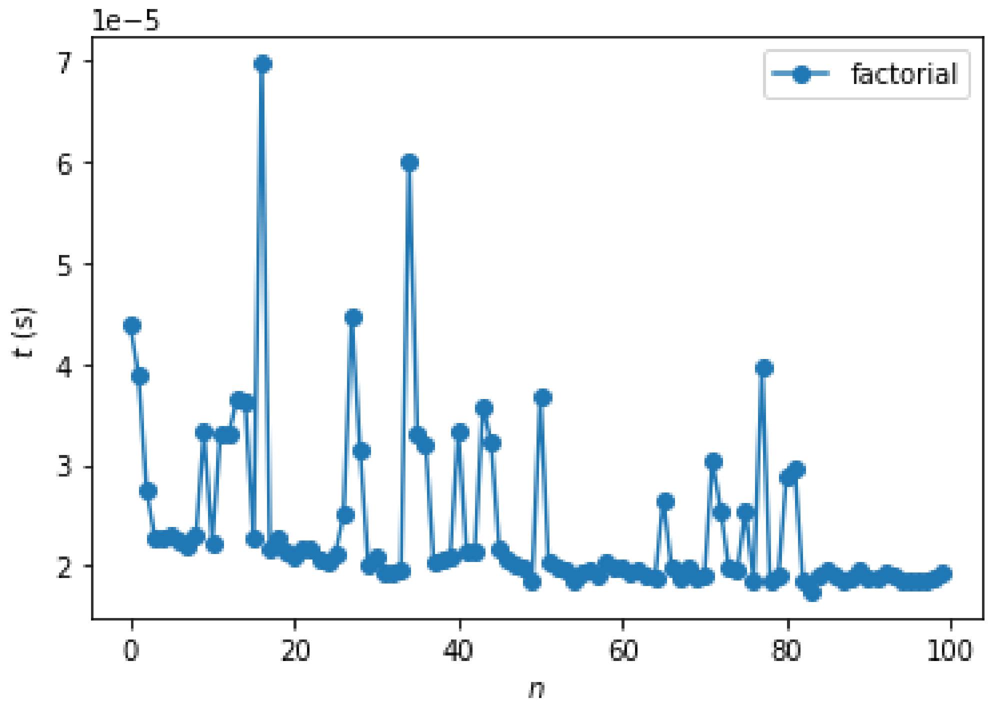
0.2.1 Solution

Recall the factorial algorithm from Activity 04.4:

[0]:
```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n - 1)*n
```

The function calls itself (recursively) n times, hence it has complexity $O(n)$. We test this below and plot the times.

[14]:
```python
# Create array of problem sizes we want to test (powers of 2)
#N = np.arange(2, 8)
# Create an array of random numbers
#x = np.random.rand(N[-1])
N = 100

# Time quicksort on arrays of different lengths
times = []
for n in range(N):
    t = %timeit -n1 -r1 -o -q factorial(N)
    times.append(t.best)

# Plot quicksort timings
plt.plot(times, marker='o', label='factorial')

# Show reference line of O(n*log(n))
#plt.loglog(N, 1e-6*N, label='$O(n)$')

# Add labels
plt.xlabel('$n$')
plt.ylabel('$t$ (s)')
plt.legend(loc=0)

plt.show()
```



0.3 Exercise 11.3

Determine experimentally the complexity of computing the determinant of a matrix. You can generate an $n \times n$ matrix using:

[0]:
```
n = 100
A = np.random.rand(n, n)
```

and the determinant can be computed by:

[0]:
```
det = np.linalg.slogdet(A)
```

Be sure that you test for sufficiently large n to get into the 'large' n regime.

0.3.1 Solution

Time computation of determinant:

[0]:
```
# Create array of problem sizes we want to test (powers of 2)
N = 2**np.arange(2, 12)
# Create an array of random numbers
x = np.random.rand(N[-1])

# Time quicksort on arrays of different lengths
times = []
```

```
for n in N:
    t = %timeit -n1 -r1 -o -q det
    times.append(t.best)
```

Plot result:

[0]:
```
# Plot quicksort timings
plt.loglog(N, times, marker='o', label='det')

# Show reference line of O(n*log(n))
plt.loglog(N, 1e-6*N, label='$O(n)$')

# Add labels
plt.xlabel('$n$')
plt.ylabel('$t$ (s)')
plt.legend(loc=0)

plt.show()
```