MA934 - Week 6 (unassessed!) Problem Sheet

The problems below give you a chance to practice some of the concepts we learned about in week 1 and also provide a chance to hand in work that would receive feedback ahead of any assessed submission. This is an opportunity to double-check both if mathematical concepts are covered sufficiently thoroughly and whether implementation elements (commenting, plot creation) are likely to result in full marks.

Task 1

Create a notebook and implement code that reads in parameters α and n and produces a log plot of some samples of the function

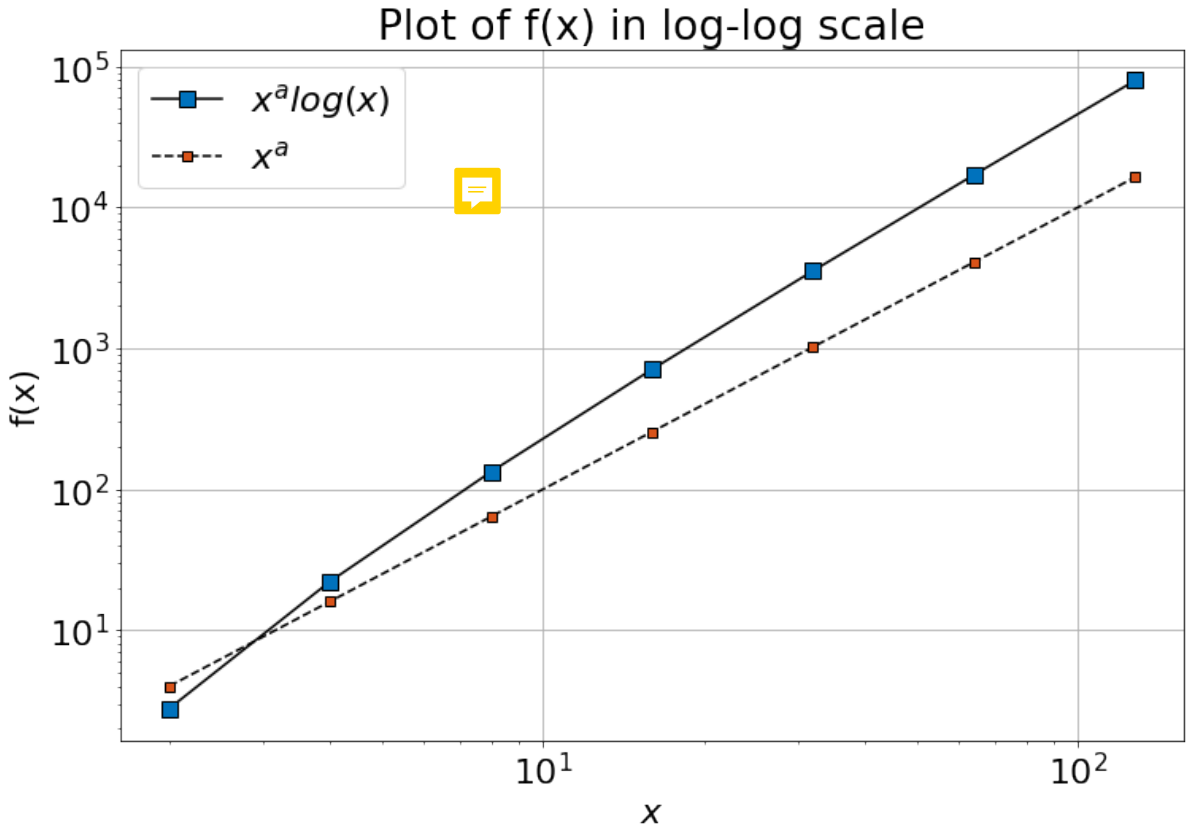
$$f(x) = x^{\alpha}\ \log(x)$$

at values of x that increase in powers of 2 from 1 to 2^n .

In [124]:
```
import numpy as np
import matplotlib.pyplot as plt

#fucntion itself
def f(x, alpha):
    return x**alpha*np.log(x)

#read alpha and n from keyboard
print("Input alpha, float:")
alpha = float(input())
print("Input alpha, int:")
n = int(input())

#make x and f(x)
mas_x = np.array([2**i for i in range(1, n+1)])
mas_fx = [f(x, alpha) for x in mas_x]

#plot the results
plt.figure(figsize=(12, 8))
plt.loglog(mas_x, mas_fx, "-ks", label='$x^a log(x)$', markersize=1
plt.loglog(mas_x, [x**alpha for x in mas_x], "--ks", label='$x^a$',
plt.xlabel('$x$')
plt.ylabel('f(x)')
plt.title('Plot of f(x) in log-log scale')
plt.legend()
plt.grid()
plt.show()
```

```
Input alpha, float:
2
Input alpha, int:
```

7



If we take the logarithms from both sides of our fuction $f(x) = x^{\alpha}\log(x)$, we will have that $\log(f(x)) = \alpha\log(x) + \log(\log(x))$, and since for large x the leading term is $\log(x)$, then the dependence of $\log f(x)$ on $\log x$ is almost linear, so in log-log scale we will see the line, whose tangent angle will be α — and that is what we see on the graph

Task 2

Following a similar structure (including a function - or more!) to the Notebook presented in class for matrix-matrix multiplication, add to the respective code such that it can also:

- execute the Strassen multiplication algorithm
- plot the runtime on the same figure as the standard matrix-matrix multiplication results and compare the two

Note: you may benefit from reducing the size of the problem while you do so (to avoid unnecessarily long execution while you debug).

In [90]:

```
n=1
A = np.random.rand(n, n)
B = np.random.rand(n, n)

# A=np.array([1,2,3,4]).reshape(2,2)
```

```
# B=np.array([5,60,7,8]).reshape(2,2)

def MMStrassen(A, B):
    # Detect matrix size
    n = len(A)

    #check whether n is the power of 2
    if (n%2 == 1 and n != 1):
        raise RuntimeError("Encountered n={}, but Strassen method w

    if (n == 1):
        return A * B

    nSize = (n, n)
    # Initialise result matrix
    C = np.zeros(nSize)



    A11 = A[0:n//2, 0:n//2]
    A12 = A[0:n//2, n//2:]
    A21 = A[n//2:, 0:n//2]
    A22 = A[n//2:, n//2:]

    B11 = B[0:n//2, 0:n//2]
    B12 = B[0:n//2, n//2:]
    B21 = B[n//2:, 0:n//2]
    B22 = B[n//2:, n//2:]

    if (n > 2):
        M1 = MMStrassen(A11 + A22, B11 + B22)
        M2 = MMStrassen(A21 + A22, B11)
        M3 = MMStrassen(A11, B12 - B22)
        M4 = MMStrassen(A22, B21 -  B11)
        M5 = MMStrassen(A11 + A12, B22)
        M6 = MMStrassen(A21 - A11, B11 + B12)
        M7 = MMStrassen(A12 - A22, B21 + B22)
    else:
        M1 = (A11 + A22) * (B11 + B22)
        M2 = (A21 + A22) * B11
        M3 = A11 * (B12 - B22)
        M4 = A22 *(B21 -  B11)
        M5 = (A11 + A12) * B22
        M6 = (A21 - A11) * (B11 + B12)
        M7 = (A12 - A22) * (B21 + B22)

    C11= M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C[0:n//2, 0:n//2] = C11
    C[0:n//2, n//2:] = C12
```

```
    C[n//2:, 0:n//2] = C21
    C[n//2:, n//2:] = C22

    return C

MMStrassen(A, B)
```

Out[90]:
```
array([[0.03027904]])
```

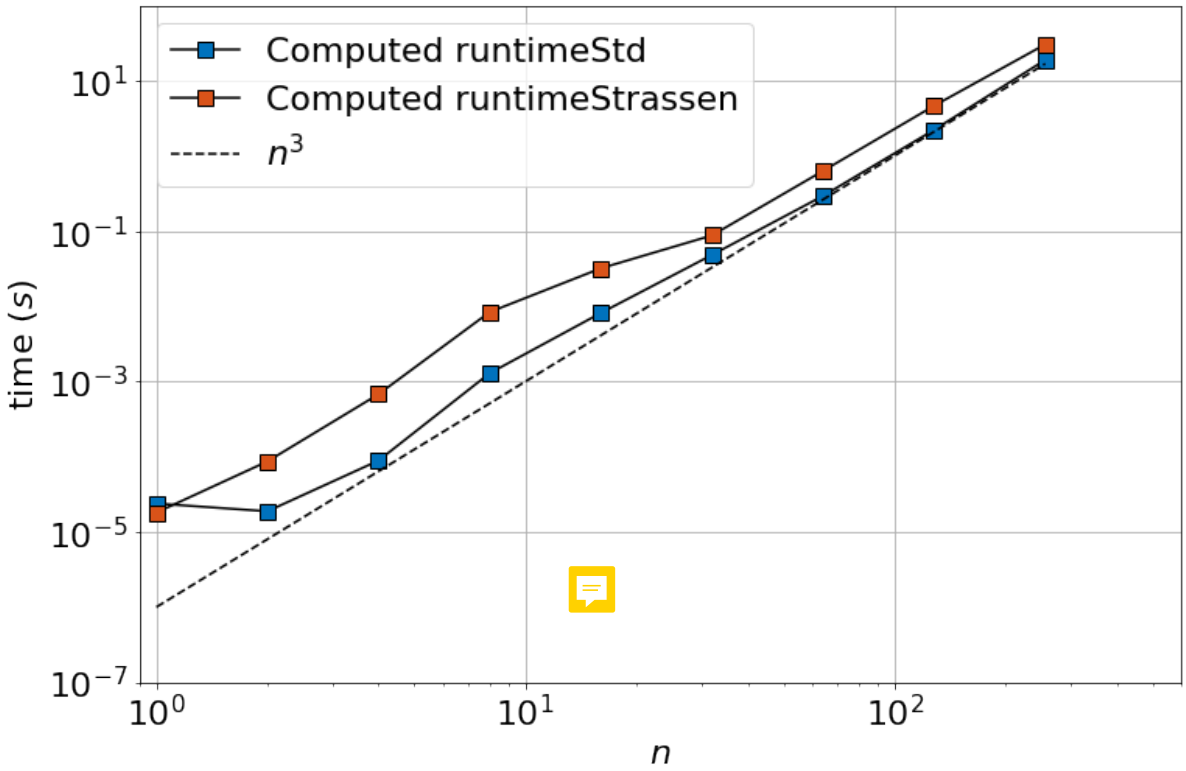
In [104]:
```
import timeit
from tqdm import tqdm

# function that computes the standard matrix-matrix multiplication
def MMStd(A, B):
    # Detect matrix size
    n = len(A)
    nSize = (n, n)
    # Initialise result matrix
    C = np.zeros(nSize)

    # Loop through the matrix elements
    for i in range(n):
        for j in range(n):
            for k in range(n):
                # Use standard multiplication formula
                C[i][j] += A[i][k]*B[k][j]
    return C

def MMStrassen(A, B):
    # Detect matrix size
    n = len(A)

    #check whether n is the power of 2
    if (n%2 == 1 and n!=1):
        raise RuntimeError("Encountered n={}, but Strassen method w

    if (n==1):
        return A * B

    nSize = (n, n)
    # Initialise result matrix
    C = np.zeros(nSize)




    A11 = A[0:n//2, 0:n//2]
    A12 = A[0:n//2, n//2:]
    A21 = A[n//2:, 0:n//2]
    A22 = A[n//2:, n//2:]

    B11 = B[0:n//2, 0:n//2]
    B12 = B[0:n//2, n//2:]
    B21 = B[n//2:, 0:n//2]
```

```
    B21 = B[n//2:, 0:n//2]
    B22 = B[n//2:, n//2:]

    if (n > 2):
        M1 = MMStrassen(A11 + A22, B11 + B22)
        M2 = MMStrassen(A21 + A22, B11)
        M3 = MMStrassen(A11, B12 - B22)
        M4 = MMStrassen(A22, B21 -  B11)
        M5 = MMStrassen(A11 + A12, B22)
        M6 = MMStrassen(A21 - A11, B11 + B12)
        M7 = MMStrassen(A12 - A22, B21 + B22)
    else:
        M1 = (A11 + A22) * (B11 + B22)
        M2 = (A21 + A22) * B11
        M3 = A11 * (B12 - B22)
        M4 = A22 *(B21 -  B11)
        M5 = (A11 + A12) * B22
        M6 = (A21 - A11) * (B11 + B12)
        M7 = (A12 - A22) * (B21 + B22)

    C11= M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C[0:n//2, 0:n//2] = C11
    C[0:n//2, n//2:] = C12
    C[n//2:, 0:n//2] = C21
    C[n//2:, n//2:] = C22

    return C

# This gives us the list of powers of two up to a specified amount
sizes = 2 ** np.arange(9)

# Initialise the list of runtimes with zeros
runtimesStd = [np.float32(0.0)] * 9
runtimesStrassen = [np.float32(0.0)] * 9

# Initialise an iterator to zero as well
j=0

for i in tqdm(sizes):
    print("j=", j)
    A = np.random.rand(i, i)
    B = np.random.rand(i, i)

    starttime = timeit.default_timer()
    CStd = MMStd(A, B)
    runtimesStd[j] = timeit.default_timer() - starttime

    starttime = timeit.default_timer()
    CStrassen = MMStrassen(A, B)
    runtimesStrassen[j] = timeit.default_timer() - starttime
    j=j+1
```

```
    J=J+1
```

```
 67%|██████▋    | 6/9 [00:00<00:00, 31.79it/s]

j= 0
j= 1
j= 2
j= 3
j= 4
j= 5
j= 6
j= 7
j= 8

100%|██████████| 9/9 [00:57<00:00,  6.35s/it]
```

In [107]:
```
plt.rcParams.update({'font.size': 22})

plt.figure(figsize=(12, 8))

plt.loglog(sizes, [i for i in runtimesStd], "-ks", label=r"Computed
plt.loglog(sizes, [i for i in runtimesStrassen], "-ks", label=r"Com
plt.loglog(sizes, 1e-6*np.power(sizes, 3.0), "--k", label=r"$n^3$")

plt.legend(loc="upper left")

plt.xlabel(r"$n$")
plt.ylabel(r"time $(s)$")
plt.xlim([0.9, 600])
plt.ylim([1e-7, 1e2])
plt.grid()
plt.show()
```



Now let's find of which power of n is the complexity of Std and Strassen method;
$y = Cn^a$ => $\log y = a \log n + \log C$ => a is the tangent

In [122]:
```
#a for Std method
logy = np.log(runtimesStd)
(logy[-1] - logy[-2])/(np.log(2**8)-np.log(2**7))
```

Out[122]:
```
3.086129901938757
```

In [121]:
```
#a for Strassen method
logy = np.log(runtimesStrassen)
(logy[-1] - logy[-2])/(np.log(2**8)-np.log(2**7))
```

Out[121]:
```
2.709169551868577
```

So we see that both powers are as expected; but note that Strassen times are greater than Std times, despite the fact that the power of n So Strassen is more effective for big n - but for small n it is less effective due to greater amortization constant in O(n)

Task 3

Write a new module containing functions that compute the n^{th} term, a_n , in the Fibonacci sequence:

- Iteratively
- Recursively
- Using memoization

The functions should work starting from any given values of a_1 and a_2 .

Measure the run-time of each of these functions over a range of values of n and produce a plot to illustrate your results.

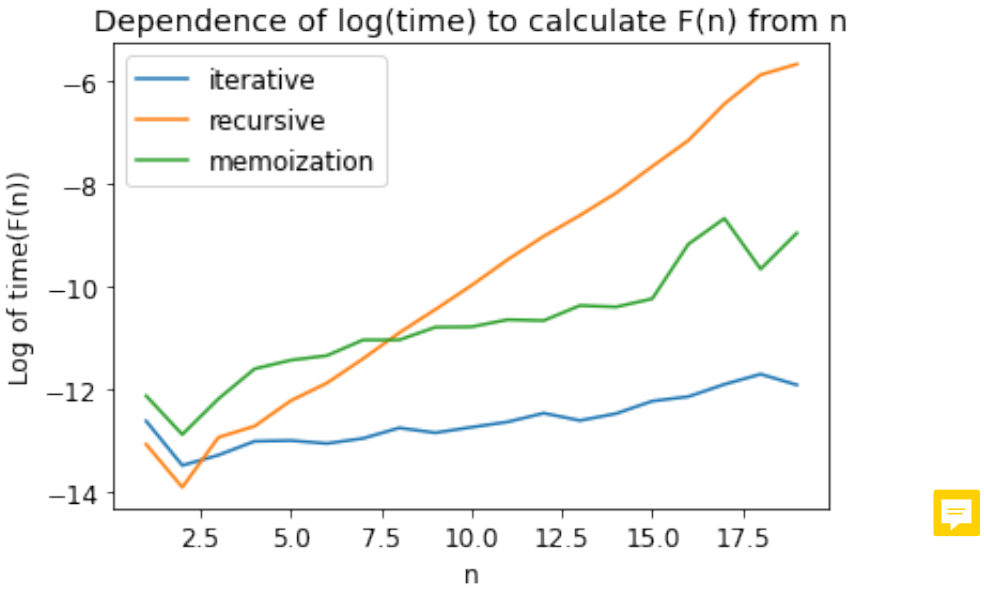
The recursive function implementation gets slow very quickly - I could only do up to $n = 48$ on my laptop. Write a recursion relation for the computational complexity of the recursive version of the task. Solve it to prove that the computational complexity grows exponentially with n.

In [172]:
```
def cached(func):
    cache = dict()
    @functools.wraps(func)
    def wrapper(*args):
        key = (func, args)
        if key not in cache:
            cache[key] = func(*args)
        return cache[key]
    return wrapper

class MyFibClass:

    def __init__(self, a1, a2):
        self.a1 = a1
        self.a2 = a2

    def get_Fn_iterative(self, n):
        if (n == 1):
            return self.a1
        if (n == 2):
```

```
            return self.a2
        else:
            anm2 = self.a1
            anm1 = self.a2
            an = anm1 + anm2
            j = 3
            while (j < n):
                anm2 = anm1
                anm1 = an
                an = anm1 + anm2
                j += 1
        return an

    def get_Fn_recursive(self, n):
        if (n == 1):
            return self.a1
        if (n == 2):
            return self.a2
        else:
            return self.get_Fn_recursive(n-1) + self.get_Fn_recursi


    @cached
    def get_Fn_memoization(self, n):
        if (n == 1):
            return self.a1
        if (n == 2):
            return self.a2
        else:
            return self.get_Fn_memoization(n-1) + self.get_Fn_memoi


times_iterative = []
times_recursive = []
times_memoization = []

nRange = range(1,20)
for n in tqdm(nRange):
    a1 = np.random.randn()
    a2 = np.random.randn()
    F=MyFibClass(a1, a2)

    starttime = timeit.default_timer()
    x = F.get_Fn_iterative(n)
    ti = timeit.default_timer() - starttime
    times_iterative.append(ti)

    starttime = timeit.default_timer()
    y = F.get_Fn_recursive(n)
    tr = timeit.default_timer() - starttime
    times_recursive.append(tr)

    starttime = timeit.default_timer()
    z = F.get_Fn_memoization(n)
```

```
    tm = timeit.default_timer() - starttime
    times_memoization.append(tm)


plt.rcParams.update({'font.size': 12})
plt.figure(figsize=(6, 4))

plt.plot(nRange, np.log(times_iterative), label='iterative')
plt.plot(nRange, np.log(times_recursive), label='recursive')
plt.plot(nRange, np.log(times_memoization), label='memoization')
plt.xlabel('n')
plt.ylabel('Log of time(F(n))')
plt.title('Dependence of log(time) to calculate F(n) from n')
plt.legend()
plt.show()
```

```
100%|██████████| 19/19 [00:00<00:00, 1633.26it/s]
```



We see that recursion needs much more time than iterative method, and that memoization improves the situation significantly

Let's write a recutsion relation for computational complexity; we know that if we have F_{n-1} and F_{n-2}, we just need to add them - it is one operation; So if we denote T_n the needed complexity to compute F_n, then $T_n = T_{n-1} + T_{n-2} + 1$ with $T_1 = 0, T_2 = 0$; Then the general solution is $T_n = c_1((1 - \sqrt{5})/2)^n + c_2((1 + \sqrt{5})/2)^n$, where C_1 and C_2 are found from the condition that $T_1 = 0, T_2 = 0$, so $C_2 = -C_1 = 1/(2\sqrt{5})$; and while the first term dies to zero as $n \to \infty$, the second term has module greater than 1, so it grows, and grows exponentially

In [ ]:

Task 4

The computational complexity of the naive divide-and-conquer algorithm for matrix multiplication satisfies the functional equation

$$F(n) = 8F(n/2) + 4(n/2)^2$$

with $F(1) = 1$.

The corresponding equation for Strassen multiplication is

$$F(n) = 7F(n/2) + 18(n/2)^2$$

with $F(1) = 1$.

Solve these recurrence relations explicitly to prove that the computational complexity of the two algorithms are $O(n^3)$ and $O(n^{\log_2(7)})$ respectively.

It is helpful to adopt the change of variables $n = 2^p$ with $a_p = F(2^p)$ to obtain linear (albeit inhomogeneous) recursion relations.

Some helpful online notes about solving recursion relations can be found at:

https://www.tutorialspoint.com/discrete_mathematics/discrete_mathematics_recurrence_re
(https://www.tutorialspoint.com/discrete_mathematics/discrete_mathematics_recurrence_r

Solve $F(n)=8F(n/2)+4(n/2)^2$

Change of veriables: $n=2^p$, $a_p=F(2^p)$

Then $F(n)=8F(n/2)+4F(n/2)^2$ turns into $F(2^p)=8F(2^{p-1}) + 4(2^{p-1})^2$,

so $a_p=8a_{p-1} + 2^{2p}$ with $a_0=1$

So characteristic equation is $\lambda=8$

So $a_{p_{homogeneous}} = C_1 8^p = C_1 2^{3p}$

And look for the solution of inhomogeneous equation in the form $a_{p_{inhomogeneous}}=C_2 4^p$

So $C_2 4^p = 8 C_2 4^{p-1} + 4^p$

So $C_2 = 2 C_2 + 1$

So $C_2=1$

So $a_p= C_1 2^{3p} - 4^p = C_1 2^{3p} - 2^{2p}$

And since $a_0=1$, we have $C_1-1=1 => C_1=2$

So $a_p=2 * 2^{3p} - 2^{2p} = 2^{2p} \left(2^{p+1} - 1 \right)$

And change variables back:

$p= \log_2(n)$ gives

$F(n)=n^2(2n-1)$

Solve $F(n) = 7F(n/2) + 18(n/2)^2$

Change of veriables: $n = 2^p$, $a_p = F(2^p)$

Then $F(n) = 7F(n/2) + 18F(n/2)^2$ turns into $F(2^p) = 7F(2^{p-1}) + 18(2^{p-1})^2$,

so $a_p = 7a_{p-1} + \frac{9}{2}2^{2p}$ with $a_0 = 1$

So characteristic equation is $\lambda = 7$

So $a_{p_{homogeneous}} = C_1 7^p$

And look for the solution of inhomogeneous equation in the form $a_{p_{inhomogeneous}} = C_2 4^p$

So $C_2 4^p = 7C_2 4^{p-1} + \frac{9}{2}4^p$

So $4C_2 = 7C_2 + 18$

So $C_2 = -6$

So $a_p = C_1 7^p - 6 \cdot 4^p$

And since $a_0 = 1$, we have $C_1 - 6 = 1 => C_1 = 7$

So $a_p = 7 \cdot 7^p - 6 \cdot 4^p$

And change variables back:

$p = \log_2(n)$ gives

$$F(n) = 7 \cdot 7^{\log_2(n)} - 6n^2 = 7 \cdot 7^{\frac{\log_7(n)}{\log_7(2)}} - 6n^2 = 7n^{\log_2(7)} - 6n^2$$

In [ ]: