1. Draw a the merger on 16 elements that you get when you use the strategy of the reading on sorting networks.

2. Draw the sorting network on sixteen elements you get from using these mergers.

3. Here is another idea for a merger. Suppose again that the number $n$ of lines is a power of two and numbered from 1 through $n$ from top to bottom. A precondition is that the 0’s and 1’s on the first $n/2$ lines come in sorted, as do the 0’s and 1’s on the last $n/2$ lines.
(a) If the last 0 on the first \( n/2 \) lines occurs on an even-numbered line, what is the number of 1’s on the first \( n/4 \) even-numbered lines minus the number of 1’s on the first \( n/4 \) odd-numbered lines? Recall that the first \( n/2 \) lines are already sorted.

(b) If the last 0 on the first \( n/2 \) lines occurs on an odd-numbered line, what is the difference between the number of 1’s on the first \( n/4 \) even-numbered lines and the first number of 1’s on the first \( n/4 \) odd-numbered lines?

(c) Over all \( n \) lines, what values can the number of 1’s on the even-numbered lines minus the number of 1’s on the odd numbered lines take on? (There are only three cases.)

(d) Suppose we correctly merge the even-numbered lines and correctly merge the odd-numbered lines. This has the effect of sorting separately the even-numbered lines and odd-numbered lines. For which of the three cases from the previous subproblem does this give a correct sort of the whole sequence?
(e) Convince yourself that for the case where it fails, the only defect is the need to swap a 1 on an even-numbered line with a 0 on the next line, which is odd. Draw the mergers you get for 2, 4, 8, and 16 elements using this strategy.

(f) Derive a recurrence for the running time, and a big-Θ time bound in closed form for your merger.

(g) Derive a recurrence for the number of comparisons it makes for a list of size $n = 2^k$ and a big-Θ bound in closed form, expressed as a function of $n$. 
4. The book measures Strassen’s time bound for multiplying $n \times n$ matrices as a function of $n$. What would have expressed it as a function of the number $m$ of scalars in the matrices, where $m = n \times n$. This would have been a natural way to do it, since it gives the time bound as a measure of the input size, which is $\Theta(m)$.

(a) Express the naive time bound for multiplication of two square as a function of $m$.

(b) Give a recurrence for the time bound of Strassen’s algorithm in terms of $m$, and use the master theorem to get a time bound in terms of $m$. Since we assume that $n$ is a power of two and $m = n^2$, assume that $m$ is a power of four.

(c) Argue in one or two sentences that the resulting time bound is equivalent to the one we have derived, just expressed in terms of $m$ instead of $n$. 
Programming

Rename your program file from Homework 2 to Hw3.py. (You may use the posted solution if you got things wrong in your program.) Add the following to the top of the file:

```python
import time
import math
import cmath
import random
import numpy as np
```

These give you access to libraries of functions for various useful operations. For example, if \( r \) is an integer, \( \text{int}(\text{np.rint}(r)) \) gives you \( r \) rounded to the nearest integer, and represented as a Python int. Try it out by running `ipython -i Hw3.py` and then playing with the following:

```plaintext
In [1]: np.rint(3.54)
Out[1]: 4.0

In [2]: int(np.rint(3.54))
Out[2]: 4
```

Here is another thing to try from these libraries:

```plaintext
In [3]: random.randint(3,100)
Out[3]: 28
```

```plaintext
In [4]: [random.randint(0,1000) for i in range(10)]
Out[4]: [144, 757, 230, 273, 209, 316, 103, 826, 976, 550]
```

This is useful for generating large random sequences to test your program on. You can find explanations in the tutorial that I posted a link to. Using Google, you can look up ways to use `time`, which allows you to time how long your implementations run on large examples.

What follows is an implementation of the FFT that, given the way I’ve explained the algorithm, is more intuitive than the text’s. (The text’s pays greater attention to minimizing constant factors in the running time, which you would want to consider in an industrial implementation.) It uses `cmath` for some complex-number operations.

```python
#Find the DFT or its inverse of a sequence given by A. If inverse == False, # find the DFT. If inverse == True, find the inverse DFT
def FFT(A, inverse):
```

5
l = nextPowerTwo(len(A))  # smallest power of two >= this degree bound
A = A + [0] * (l - len(A))
if inverse == False:
    # e^{2pi/len(A)} gives omega_n ...
    omega_n = cmath.exp(complex(0, 2*cmath.pi/len(A)))
else:
    # if the inverse, go the opposite way around the unit circle ...
    omega_n = cmath.exp(complex(0, -2*cmath.pi/len(A)))
Y = FFTAux(A, omega_n)
if inverse == False:
    return Y
else:
    return [Y[i] / len(Y) for i in range(len(Y))]

def FFTAux(A, omega_n):
    if len(A) == 1: return A
    else:
        AE = A[0:len(A):2]
        onsq = omega_n * omega_n
        YE = FFTAux(AE, onsq)
        YO = FFTAux(AO, onsq)
        YE = YE + YE
        YO = YO + YO
        omega = 1.0
        Result = [0]* len(A)
        for i in range(len(A)):
            Result[i] = (YE[i] + YO[i] * omega)
            omega = omega * omega_n
        return Result

1. Introduce the above to your file and try it out. Then make a copy of these two methods, and adapt them so that they compute the DFT for \( n \) a power of three instead of a power of two. `FFTAux` should make three recursive calls, rather than two.

2. Introduce `convolve4`, which works just like `convolve2`, but that calls the version of the FFT I wrote for you. Its running time is \( O(n \log n) \). To make sure you’ve done it right, try it on large randomly generated sequences and verify that it is dramatically faster than the naive method.