Consider a signal \( x_1[n] = (-j)^n + e^{j\pi n/4} \)

Compute the DFT coefficients with a sensible value of \( N \). What value of \( N \) do you want to use for this analysis, and why? How would your answer change if you used \( N' = 2N \) instead?

\[
\begin{align*}
x_1[n] &= (-j)^n + e^{j\pi n/4} = e^{\frac{3\pi}{2}n} + e^{\frac{\pi}{4}n}
\end{align*}
\]

This signal is periodic in 8 so it is reasonable to choose a window of \( N = 8 \).

In this view, we already have \( x_1 \) expressed as a sum of complex exponentials, one of which goes through one cycle in our analysis, and another which goes through 6 cycles in our analysis window (-2 in the baseband). So we should have two non-zero coefficients, one at \( k = 6 \) (or \( k = -2 \)), and another at \( k = 1 \). Since the amplitudes of both waves are 1, these values should both be exactly 1.

Computing directly using the DFT analysis equation:

\[
\begin{align*}
X_1[k] &= \sum_{n=0}^{7} x_1[n]e^{-j\frac{2\pi}{8}kn} \\
        &= \sum_{n=0}^{7} e^{-j\frac{\pi}{4}n(6-k)} + e^{-j\frac{\pi}{4}n(1-k)} \\
        &= \delta[k - 6] + \delta[k - 1]
\end{align*}
\]

If we double the window, \( N' = 2N = 16 \):

\[
\begin{align*}
X_1[k] &= \sum_{n=0}^{15} x_1[n]e^{-j\frac{2\pi}{16}kn} \\
        &= \sum_{n=0}^{15} e^{-j\frac{\pi}{8}n(12-k)} + e^{-j\frac{\pi}{8}n(2-k)} \\
        &= \delta[k - 12] + \delta[k - 2]
\end{align*}
\]

(The DFT coefficients with the value of 1 change indices to \( k = 2 \) and \( k = 12 \) (\( k = -4 \)).)
Part 2.

Consider a different signal $x_2[n]$ that is known to be zero outside the range $0 \leq n < 6$, whose corresponding DFT (computed with $N = 6$) is given by:

$$X_2[k] = X_2[k + 6] = \begin{cases} 1 & \text{if } k = 0 \\ j & \text{if } k = 2 \\ -j & \text{if } k = 4 \\ 0 & \text{otherwise} \end{cases}$$

Consider a new signal $y_1[n] = 9 - 2x_2[n]$. What are the DFT coefficients of $y_1$, $Y_1[k]$? Explain how to arrive at your solution without explicitly determining $x_2[n]$.

By linearity, $Y_1[k] = V[k] - 2X_2[k]$, where $V[k]$ represents the DFT coefficients associated with the signal $v[n] = 9$.

$$V[k] = \frac{1}{6} \sum_{n=0}^{5} 9e^{j \frac{2\pi kn}{6}} = 9\delta[k]$$

Therefore,

$$Y_1[k] = Y_1[k + 6] = 9\delta[k] - 2X_2[k] = \begin{cases} 7 & \text{if } k = 0 \\ -2j & \text{if } k = 2 \\ 2j & \text{if } k = 4 \\ 0 & \text{otherwise} \end{cases}$$

Consider another new signal $y_2[n] = 5(-1)^nx_2[n]$. What are the DFT coefficients of $y_2$, $Y_2[k]$? Explain how to arrive at your solution without explicitly determining $x_2[n]$.

Scaling the time-domain representation by 5 corresponds to scaling the frequency-domain representation by 5.

Re-expressing $(-1)^n$ as $e^{j\pi n}$, we see that this factor introduces a linear phase term in the time domain. Just as a phase shift of this form in the frequency domain corresponded to a time shift in the time domain, a phase shift of this form in the time domain corresponds to a frequency shift in the frequency domain. In particular, this shifts the coefficients by 3, so we have:

$$Y_2[k] = 5X_2[k - 3]$$

If one period of $X_2$ was $\{1, 0, j, 0, -j, 0\}$, then one period of $Y_2$ is $\{0, -5j, 0, 5, 0, 5j\}$. 
Part 3.

Consider a signal $x_3[n]$, which is known to be nonzero only for $n = 0, 1, 2, 3$.

If you know the values of the DTFT at two points only (in particular, you know the values $X_3(\frac{\pi}{2})$ and $X_3(\frac{3\pi}{2})$). You also know the value of $\sum_m x_3[m]$ and $\sum_m e^{j\pi m} x_3[m]$.

Explain how you could determine $x_3[n]$ from this information.

$x_3[n]$ is non-zero for $n = 0, 1, 2, 3$. The DTFT of $x_3[n]$ is:

$$X_3(\Omega) = \sum_{n=-\infty}^{\infty} x_3[n] e^{-j\Omega n}$$

$$= \sum_{n=0}^{3} x_3[n] e^{-j\Omega n}$$

Let $X_3[K]$ be the DFT of $x_3[n]$ with analysis window $N = 4$. Then,

$$X_3[0] = \frac{1}{4} \sum_{n=0}^{3} x_3[n] e^{-j\frac{2\pi}{4} n 0} = \frac{1}{4} \sum_{n=0}^{3} x_3[n]$$

$$X_3[1] = \frac{1}{4} \sum_{n=0}^{3} x_3[n] e^{-j\frac{2\pi}{4} n 1} = \frac{1}{4} X_3\left(\frac{\pi}{2}\right)$$

$$X_3[2] = \frac{1}{4} \sum_{n=0}^{3} x_3[n] e^{-j\frac{2\pi}{4} n 2} = \frac{1}{4} \sum_{n=0}^{3} x_3[n] e^{-j\pi n} = \frac{1}{4} \sum_{n=0}^{3} x_3[n] e^{j\pi n}$$

$$X_3[1] = \frac{1}{4} \sum_{n=0}^{3} x_3[n] e^{-j\frac{2\pi}{4} n 3} = \frac{1}{4} X_3\left(\frac{3\pi}{2}\right)$$

The right hand side of each of the equations are given as known in the problem. We can determine $x_3[n]$ by applying the DFT synthesis equation using these four values.
2 Real World: Cambridge

Part 1.

Ben Bitdiddle notices that, in the DFT of a purely real signal computed with a value of \( N \) that is even, two coefficients in particular (namely \( k = 0 \) and \( k = N/2 \)) are always real-valued (whereas other values can be complex). Why is this the case? Prove by expressing those two DFT coefficients in a form that demonstrates that they are always real-valued.

When analyzing with an odd-valued \( N \), are there any coefficients that are similarly guaranteed to be real? Explain.

In general, we have:

\[
X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{j2\pi kn/N}
\]

For \( k = 0 \), this becomes:

\[
X[0] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{0} = \frac{1}{N} \sum_{n=0}^{N-1} x[n]
\]

If \( x[n] \) is real-valued for all \( n \), then this sum must be real-valued.

Similarly, when \( k = N/2 \), we have:

\[
X[N/2] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{j\pi n} = \frac{1}{N} \sum_{n=0}^{N-1} x[n](-1)^n
\]

If \( x[n] \) is real-valued for all \( n \), then this sum must be real-valued.

This will always happen when the \( \Omega \) value associated with a certain \( k \) is 0 or \( \pi \) (since those are the only \( \Omega \) values for which \( e^{j\Omega n} \) will always be real-valued). If \( N \) is odd-valued, we have \( \Omega = 0 \) represented, but we do not have \( \pi \) represented, so the only value that is guaranteed to be real-valued is \( k = 0 \) (which is associated with \( \Omega = 0 \)).
Part 2.

Ben’s friend Lem E. Tweakit notices that computing the $N$-point DFT of a purely real signal $x[n]$ is taking a long time, when he gets an idea: he constructs a new signal $x'[n]$, defined as follows:

$$x'[n] = x[n] + jx[n + N/2]$$

He then computes a $(N/2)$-point DFT of the resulting (complex-valued) signal and finds the coefficients $X'[k]$. As he suspected, this process takes a lot less time. However, Ben thinks that Lem’s strategy has resulted in meaningless coefficients.

Help them resolve their conflict! If we were given only $X'[k]$, and we were told how it was computed, could we reconstruct the original signal $x[n]$? If so, how? If not, why not?

The intention of this problem was to find $x[n]$ without explicitly solving for $x'[n]$. We’ll take that approach here.

Some key observations we have used earlier in 6.003:

- A purely real and even signal has DFT coefficients that are real and even.
- A purely real and odd signal has DFT coefficients that are imaginary and odd.

Thus, we can say that, for an arbitrary real-valued signal in time, the even part of that signal will affect the real part of the DFT coefficients (which will be even), and the odd part will affect the imaginary part of the DFT coefficients (which will be odd).

Similarly, if we had started with a purely imaginary signal, the even part of that signal would affect the imaginary part of the DFT coefficients (which will be even), and the odd part would affect the real part of the DFT coefficients (which will be odd).

This informs our decision about how to reconstruct $x[n]$. In particular, we have:

$$\text{Re} (x'[n]) \overset{\text{DFT}}{=} \text{Ev} (\text{Re} (X'[k])) + j\text{Odd} (\text{Im} (X'[k]))$$

and

$$j\text{Im} (x'[n]) \overset{\text{DFT}}{=} j\text{Ev} (\text{Im} (X'[k])) + \text{Odd} (\text{Re} (X'[k]))$$

We can reconstruct $x[n]$ by using the two formulas above to compute $\text{Re}(x'[n])$ and $\text{Im}(x'[n])$, which correspond to the first and second half of $x[n]$, respectively.
3 Spectrogram Matching

Recall that spectrograms are computed according to three parameters: Below are several spectrograms of the same piece of music, computed with different parameters.

1. The window size $N$
2. The step size $s$, defined as a fraction of $N$
3. The windowing function

Each pair of spectrograms below differs by exactly one parameter. For each pair of spectrograms, indicate which of the following changes was made between computing the first and the second spectrogram, and briefly explain how you arrived at that conclusion. The possible changes are:

- No change was made.
- $N$ was increased or decreased.
- $s$ was increased or decreased.
- The window function was changed from rectangular to Hann, or from Hann to rectangular.

Part 1.

![Spectrogram 1.A](image1.png) ![Spectrogram 1.B](image2.png)

The frequency resolution is better in the second picture than in the first. This corresponds to increasing $N$. 
Part 2.

The left picture looks more “pixelated” in the horizontal direction, and the picture on the right looks smoother in that direction (i.e., the second has better time resolution than the first). There are two things that could have caused this behavior: decreasing $N$ or decreasing $s$. However, decreasing $N$ would also have decreased the frequency resolution. Since the frequency resolution appears not to have been changed between the two pictures, the answer must be that we decreased $s$.

Part 3.

The frequency resolution is better in the second picture than in the first. This corresponds to increasing $N$. 
Part 4.

Here, the vertical (frequency) resolution appears to be the same in the two images, so $N$ must not have changed. The increased pixellation along the horizontal axis (decreased time resolution) corresponds to increasing $s$.

Part 5.

Here, we see that the time resolution has increased from the first image to the second. This corresponds either to decreasing $N$ or decreasing $s$. Here, we also see the frequency resolution decrease from the first to the second, so we must have done this by decreasing $N$. 
In these two images, the time and frequency resolutions look roughly the same, but we see much more vertical “blurring” in the second image, compared to the first. This smearing is characteristic of a rectangular window (versus a Hann window in the left image).
4 Gotta Go Fast

Download the zip archive at https://sigproc.mit.edu/_static/fall18/pset/07/fft.zip and unzip it so you have a local copy of the files contained therein.

This question will proceed in several steps. Questions written in **bold** should be answered in your writeup, and you should also include your Python code in your submission to this problem.

You should not use any imports other than **math**. (numpy is not allowed).

**Part 1.**

The file fft.py contains an implementation of the DFT analysis equation as a function called **dft_a**. This implementation explicitly computes the sum defined in the DFT analysis equation:

\[ X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-j2\pi nk/N} \]

for \(0 \leq k < N\).

But, there is a problem with this approach. Computing the DFT for a 4096-sample waveform using this code takes maybe half a minute on most computers. For 8192 samples, it takes a couple of minutes. For 16384 samples (which corresponds to around 1/4 of a second of audio at a normal sampling rate), it would take around 10 minutes! As such, it is not terribly practical, particularly when we think about Fourier analysis of real-world signals.

The DFT is a very useful tool, but in computational settings, it is obviously more useful if we have an efficient way to compute it! Even half a minute is a long time to wait, so let’s spend some time exploring ways that we can improve on the efficiency of our programs for computing the DFT.

Let’s call the above algorithm (the direct application of the formula) **algorithm A**.

**Using algorithm A, if an initial run of the code takes \(t\) seconds to run, how long do we expect it to take if we double the size of its input? Justify your answer.**

We would expect the second run to take \(4t\) seconds. For every value of \(n\), we perform 4 multiplications, 1 exponentiation, and 1 division for every value of \(k\) (as well as \(N-1\) additions to combine them together, and 1 more division to account for the scaling factor). Since we have \(N\) total \(n\) values and \(N\) total \(k\) values, we would expect the total number of operations to be something like:

\[ \sum \text{add, mul, exp, div for every } n, k \text{ pair} + \frac{N}{2} \text{ division once per } n \]

Ultimately, as \(N\) grows, the \(N^2\) term will dominate, so we would expect the time to scale roughly quadratically with the length of the input.
Part 2.
Recall that the DFT coefficients of any real-valued signal, \( X[k] = X^*[-k] \). If we assume our input is real-valued, we can compute the DFT much more quickly! Using this idea, implement a function `dft_b` (hint: it should be possible to achieve roughly a factor of 2 improvement in speed). Let’s call this approach **algorithm B**.

Once you have implemented this function, it is worth doing a bit of testing to make sure that:

1. `dft_b` is working as expected, and
2. `dft_b` is more efficient, by roughly the factor we expect.

We can check for correctness by comparing the results of `dft_a` and `dft_b` for a few different test inputs. Ideally, the two should produce exactly the same results. In practice, there are limitations on how precisely decimal numbers can be represented in computers that can lead to small differences. So rather than checking for the exact equality of the outputs from the two functions, we should check that they are actually trying to produce the same result, but differ only in terms of rounding.

One common way to implement this is to check whether the numbers in the outputs are “close enough” to each other. If the values produced by `dft_a` are \( a_k \) and the numbers produced by `dft_b` are \( b_k \), then we might consider the two to have produced the same result if, for all \( k \), \( |a_k - b_k| < \epsilon \) for some small value of \( \epsilon \) (maybe \( 10^{-9} \) or so).

Add some code to your Python file to convince yourself that the two functions produce the same results for some number of sample inputs.

As your test inputs, you may wish to use input signals whose DFT coefficients are easy to compare/verify by hand, such as pure sinusoids. It is also a good idea to use relatively short signals for now, so that you don’t have to wait a long time to see your results.

We’re also interested in characterizing the difference in speed between these two algorithms. For a few differently-sized inputs, try running both functions on the same input and comparing the time they take (try a 512-sample input, a 1024-sample input, and a 2048-sample input). Note that you can print an estimate of the amount of time something takes with code similar to the following:

```python
import time

start_time = time.time()
# ... perform a task here
print('That task took', time.time() - start_time, 'seconds')
```

Discuss the results of this timing in your writeup, including an explanation.

What would happen if we tried applying this equation to a complex-valued signal? Would we get the right results? Why or why not?
The approach we suggested was to leverage the fact that, for a real-valued signal, the DFT coefficients are conjugate symmetric about zero, i.e., $X[k] = X^*[-k]$.

To use this, we can simply compute the first $N/2$ coefficients as in algorithm A, and then use the complex conjugates for the second half of the coefficients.

This breaks down in the case where the input signal is complex-valued, since we can’t make the assumption about conjugate symmetry in that case.
Part 3.

Our approach above managed a factor of 2 improvement in terms of speed, which is pretty good! However, the running time of our algorithm is still quadratic in the size of the input, so it is still not particularly desirable for long signals. For example, if we wanted to run this new algorithm on multiple seconds of an audio file, it would probably still take a prohibitively long time to run!

In this section, we’ll implement a variant of the Cooley-Tukey Fast Fourier Transform (FFT) Algorithm, which is able to further exploit symmetries within the analysis equation to drastically improve the speed.

Let’s begin by again looking at our analysis equation:

\[ X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi nk}{N}} \]

The technique underlying the approach comes from breaking the sum into two pieces, one over the even-indexed values of \( n \) and one over the odd-indexed values:

\[
X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi nk}{N}} = \frac{1}{N} \left( \sum_{m=0}^{N/2-1} x[2m]e^{-j\frac{2\pi (2m)k}{N}} + \sum_{m=0}^{N/2-1} x[2m + 1]e^{-j\frac{2\pi (2m+1)k}{N}} \right)
\]

Factoring out a common constant factor from the sum on the right yields:

\[
X[k] = \frac{1}{N} \left( \sum_{m=0}^{N/2-1} x[2m]e^{-j\frac{2\pi (2m)k}{N}} + e^{-j\frac{2\pi k}{N}} \sum_{m=0}^{N/2-1} x[2m + 1]e^{-j\frac{2\pi (2m+1)k}{N}} \right)
\]

And a slight rearrangement gives:

\[
X[k] = A \left( \frac{1}{N/2} \sum_{m=0}^{N/2-1} x[2m]e^{-j\frac{2\pi mk}{(N/2)}} + e^{-j\frac{2\pi k}{N}} \frac{1}{N/2} \sum_{m=0}^{N/2-1} x[2m + 1]e^{-j\frac{2\pi mk}{(N/2)}} \right)
\]

What value of \( A \) is necessary to make the results of this equation the same as the results from our original form? Why?

In order to end up with the right scaling factor, we need \( A = \frac{1}{2} \).

The key insight at this point is that each of the sums in that equation is identical to the DFT analysis equation on a different set of numbers:

\[
X[k] = A \left( \frac{1}{N/2} \sum_{m=0}^{N/2-1} x[2m]e^{-j\frac{2\pi mk}{(N/2)}} + e^{-j\frac{2\pi k}{N}} \frac{1}{N/2} \sum_{m=0}^{N/2-1} x[2m + 1]e^{-j\frac{2\pi mk}{(N/2)}} \right)
\]

DFT coefficient from even-indexed values

DFT coefficient from odd-indexed values
This arrangement reveals that we can compute the DFT coefficients of a sequence $x[n]$ by:

1. Computing the DFT of a new signal comprised of only even-indexed values from $x$ ($x[n]$ for $n = 0, 2, 4, \ldots$).
2. Computing the DFT of a new signal comprised of only odd-indexed values from $x$ ($x[n]$ for $n = 1, 3, 5, \ldots$).
3. Combining the results from steps 1 and 2 according to the equation above.

Note that this is a recursive definition, so steps 1 and 2 can be computed by following the same process.

(As is generally the case with a recursive algorithm, we’ll need to specify a base case. Since our normal approach involves solving subproblems of size $N/2$, the case where $N = 1$ makes a nice base case. What are the Fourier coefficients for a sequence where $N = 1$?)

Each of our recursive calls yields $N/2$ DFT coefficients (for each of the smaller signals), but we need to combine those into $N$ Fourier coefficients (for the original signal). Explain your strategy for this combination, including an explanation of any properties of the DFT coefficients that you are exploiting.

The key property here is that the DFT coefficients are always periodic in the length of the analysis window.

In this case, when we do the two smaller DFT’s with analysis windows of $N/2$, we then have $X_e[k] = X_e[k + N/2]$ (and same for the odd coefficients). So looking at $N$ coefficients from each of those smaller transforms, we’ll see the coefficients repeat themselves twice.

Implement this algorithm as a recursive function `dft_c`. A couple of good test cases to use while debugging might be:

1. A sequence of length 1 (make sure the base case is correct)
2. A short sequence $[1, 0, 0, 0]$ (whose Fourier coefficients we can find by hand)
3. A slightly longer sequence of the same form
4. A pure sine or cosine wave

It is worth noting that this form of the algorithm will only work when $N$ is a power of 2, since we always rely on solving subproblems of size $N/2$. So make sure that your tests cases all have an appropriate length\(^1\).

\(^1\) This is completely optional, but if you are interested to extend your implementation to work efficiently for arbitrary sizes, you can try your hand at implementing something like Bluestein’s algorithm.
Part 4.

Once again, it is worth doing a bit of testing to make sure that `dft_c` produces the same results as the original functions.

Add some code to your Python file to convince yourself that `dft_c` produces the same results as the other versions, for some number of sample inputs.

It will be really interesting to compare the speed difference between these three algorithms. For a few differently-sized inputs, try running all three of your functions on the same input and comparing the time they take (try a 512-sample input, a 1024-sample input, a 2048-sample input, and a 4096-sample input).

You probably noticed a humongous difference in speed between the three algorithms! What is it about the third algorithm that makes it so fast?

Here are some timing results (from a version of algorithm C that is not particularly well optimized):

```
times (in seconds) for 1024 samples:
Algorithm A: 1.103555679321289
Algorithm B: 0.552323579788208
Algorithm C: 0.01724100112915039

times (in seconds) for 2048 samples:
Algorithm A: 4.442674875259399
Algorithm B: 2.210524559020996
Algorithm C: 0.03651857376098633

times (in seconds) for 4096 samples:
Algorithm A: 18.448439359664917
Algorithm B: 9.14447803497314
Algorithm C: 0.07415032386779785

times (in seconds) for 8192 samples:
Algorithm A: 71.95885443687439
Algorithm B: 35.18166923522949
Algorithm C: 0.1318361759185791
```

The results are almost exactly what we expect: every doubling of the input size almost exactly quadruples the running time of algorithm A. Algorithm B always takes almost exactly half as long as algorithm A. Algorithm C is dramatically faster than either of the earlier algorithms.
One key aspect of algorithm C is that it splits a single large DFT into two smaller ones. It turns out that even if we had used the brute-force algorithm for solving the subproblems, solving two \((N/2)\)-point DFTs is dramatically faster than solving a single \(N\)-point DFT (because our brute-force algorithm scaled quadratically, we would expect solving the two smaller DFTs to take roughly half as long as solving a single DFT).

But when we are able to solve even those subproblems recursively, we are able to do the computation much faster.

If we start with an \(N\)-point DFT, we will solve two \((N/2)\)-point DFTs. But we actually solve those by breaking them down into a total of four \((N/4)\)-point DFTs. And we actually solve those by breaking them into a total of eight \((N/8)\)-point DFTs. We continue this until we reach DFTs of length 1.

Ultimately, getting to that point required us to “split” the input a total of \(\log_2 N\) times. And the beautiful thing is that combining the results of the two smaller computations at each “layer” takes a number of operations that is linear in \(N\).

So all-in-all, rather than taking time that scales like \(N^2\), this new algorithm takes time that scales like \(N \log_2 N\). So in general, it is faster by roughly a factor of \(\log_2 N/N\), which is consistent with our timing results above (for \(N = 8192\), the factor we expect to see is \(\frac{\log_2(8192)}{8192} = \frac{13}{8192} \approx .0015\), which is very close to the observed running times of algorithms A and C on the input of size 8192.

We can now perform analysis is pretty quickly, but it would be nice to be able to do synthesis efficiently as well. Use this idea to make a version of the inverse DFT (i.e., the DFT synthesis equation) that is also fast. Explain, and include your Python code along with your writeup.

The synthesis equation differs from the analysis equation in only two ways: the scaling factor, and the sign of the exponent in the complex exponential.

As such, we can produce a similarly fast version of the inverse DFT by changing the signs of all the exponents and removing the scaling factor of \(\frac{1}{2}\).