6.300: Signal Processing

Fast Fourier Transform

November 12, 2026

Fast Fourier Transform

The Fast-Fourier Transform (FFT) is an algorithm (actually a family of algorithms) for computing the Discrete Fourier Transform (DFT).

Both elegant and useful, the FFT algorithm is arguably
the most important algorithm in modern signal processing.

– **computer science** (divide-and-conquer: **elegant** and **efficient**)
– **elegant mathematics** (using alternative representations for polynomials)
– **widely used** in engineering and science (enormous practical value)

Fast Fourier Transform

The Fast-Fourier Transform (FFT) is an algorithm (actually a family of algorithms) for computing the Discrete Fourier Transform (DFT).

Both elegant and useful, the FFT algorithm is arguably
the most important algorithm in modern signal processing.

– **computer science** (divide-and-conquer: **elegant** and **efficient**)
– **elegant mathematics** (using alternative representations for polynomials)
– **widely used** in engineering and science (enormous practical value)

It's also fascinating from an **historical** perspective.

Modern interest stems most directly from James Cooley (IBM) and John Tukey (Princeton): "An Algorithm for the Machine Calculation of Complex Fourier Series," published in *Mathematics of Computation* 19: 297-301 (1965).

However there were a number previous, independent discoveries, including Danielson and Lanczos (1942), Runge and König (1924), and most significantly work by Gauss (1805).[1]

[1] http://nonagon.org/ExLibris/gauss-fast-fourier-transform

Historical Perspective

Gauss developed the basic idea behind the FFT algorithm in his study of the orbit of the then recently discovered asteroid Pallas.

Gauss' data: "declination" X (minutes of arc) v. "ascension" θ (degrees)[2]

θ:	0	30	60	90	120	150	180	210	240	270	300	330
X:	408	89	−66	10	338	807	1238	1511	1583	1462	1183	804

Fitting function:

$$X = f(\theta) = a_0 + \sum_{k=1}^{5} \left[a_k \cos\left(\frac{2\pi k\theta}{360}\right) + b_k \sin\left(\frac{2\pi k\theta}{360}\right) \right] + a_6 \cos\left(\frac{12\pi\theta}{360}\right)$$

Resulting coefficients:

k :	0	1	2	3	4	5	6
a_k :	780.6	−411.0	43.4	−4.3	−1.1	0.3	0.1
b_k :	–	−720.2	−2.2	5.5	−1.0	−0.3	–

[2] B. Osgood, "The Fourier Transform and its Applications"

Historical Perspective

Gauss developed the basic idea behind the FFT algorithm in his study of the orbit of the then recently discovered asteroid Pallas.

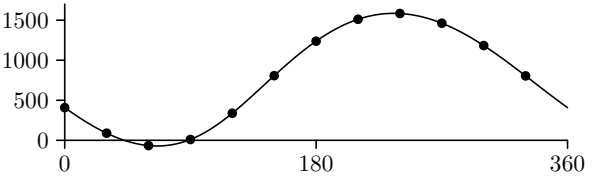
Gauss' data: "declination" X (minutes of arc) v. "ascension" θ (degrees)[3]

θ:	0	30	60	90	120	150	180	210	240	270	300	330
X:	408	89	−66	10	338	807	1238	1511	1583	1462	1183	804

Fitting function:

$$X = f(\theta) = a_0 + \sum_{k=1}^{5} \left[a_k \cos\left(\frac{2\pi k\theta}{360}\right) + b_k \sin\left(\frac{2\pi k\theta}{360}\right) \right] + a_6 \cos\left(\frac{12\pi\theta}{360}\right)$$

Resulting fit:



[3] B. Osgood, "The Fourier Transform and its Applications"

Historical Perspective

Gauss developed the basic idea behind the FFT algorithm in his study of the orbit of the then recently discovered asteroid Pallas.

Today, we think of the FFT as a **computationally efficient algorithm** (which it is). But Gauss was not so interested in computational efficiency: solving the Pallas problem involved just 12 equations in 12 unknowns!

Gauss was more interested in the **algorithmic structure** of the computation, which allowed him to rearrange the computations into a sequence of simpler problems that could then be combined to produce the final answer.

Gauss did not even publish the algorithm. The manuscript was written circa 1805 and published posthumously in 1866.

From the DFT to the FFT: Divide-and-Conquer

Key idea: the DFT can be broken into **meaningful parts** that can be solved **independently** and then combined to produce bigger subparts $\rightarrow$ with significant improvements in **speed of computation**.

Start by understanding the direct-form implementation.

Computing the DFT

Direct-form computation of DFT in Python.

$$F[k] = \frac{1}{N} \sum_{n=0}^{N-1} f[n] e^{-j\frac{2\pi kn}{N}}$$

Simple (naive) Python implementation:

```
from math import e,pi
def DFT(f):
    N = len(f)
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*e**(-2j*pi*k*n/N)/N
        F.append(ans)
    return F
```

How many operations are required by this algorithm if $N = 1024$?

1. less than 10,000
2. between 10,000 and 100,000
3. between 100,000 and 1,000,000
4. greater than 1,000,000

How does the number of operations **scale** with N?

Computing the DFT

How long does it take to compute a DFT of length N?

```
from math import e,pi
def DFT(f):
    N = len(f)
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*e**(-2j*pi*k*n/N)/N
        F.append(ans)
    return F
```

Empirical results (for my laptop):

N	seconds
1024	0.41
2048	1.67
4096	6.70
8192	27.34

Ten seconds of audio sampled at $44,100$ samples / sec $= 441,000$ samples.
Then N=$441,000 \rightarrow 0.41*(441,000/1024)^2 = 76043$ seconds $= 0.88$ days.
Ten minutes of such audio $\rightarrow 8.7$ years.

Computing the DFT

Much of the direct-form computation is in computing the **kernel functions**.

```
from math import e,pi
def DFT(f):
    N = len(f)
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*e**(-2j*pi*k*n/N)/N
        F.append(ans)
    return F
```

Computing the DFT

Much of the direct-form computation is in computing the **kernel functions**.

Complex exponentials $e^{j\theta}$ are periodic in θ with period 2π.

N unique values $\rightarrow$ **precompute** all of them!

```
from math import e,pi
def DFTprecompute(f):
    N = len(f)
    bases = [e**(-2j*pi*m/N)/N for m in range(N)]
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*bases[k*n%N]
        F.append(ans)
    return F
```

N	direct (sec.)	pre-computing
1024	0.41	0.13
2048	1.67	0.54
4096	6.70	2.15
8192	27.34	9.01

Pre-computing kernel functions reduces run-time more than a **factor of 3**.

Check Yourself

The direct-form implementation of the DFT works not only for real-valued input signals but also for complex-valued input signals.

```
from math import e,pi
def DFT(f):
    N = len(f)
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*e**(-2j*pi*k*n/N)/N
        F.append(ans)
    return F
```

How could we change the algorithm to use fewer computations for **real-valued inputs**? How many fewer computations would then be required?

Computing the DFT

The optimizations that we have discussed so far reduce computation time by a (roughly) constant factor.

While a 3-fold reduction in compution is good: 8.7 years $\rightarrow$ 2.9 years, and a 6-fold reduction is even better: 8.7 years $\rightarrow$ 1.45 years. this computation is still prohibitally slow for a 10-minute audio piece.

To reduce the number of computations more drastically, we need to reduce the order from $O(N^2)$ to a lower order – which is what the FFT algorithm does.

FFT Algorithm

Compute contributions of even and odd numbered input samples separately.

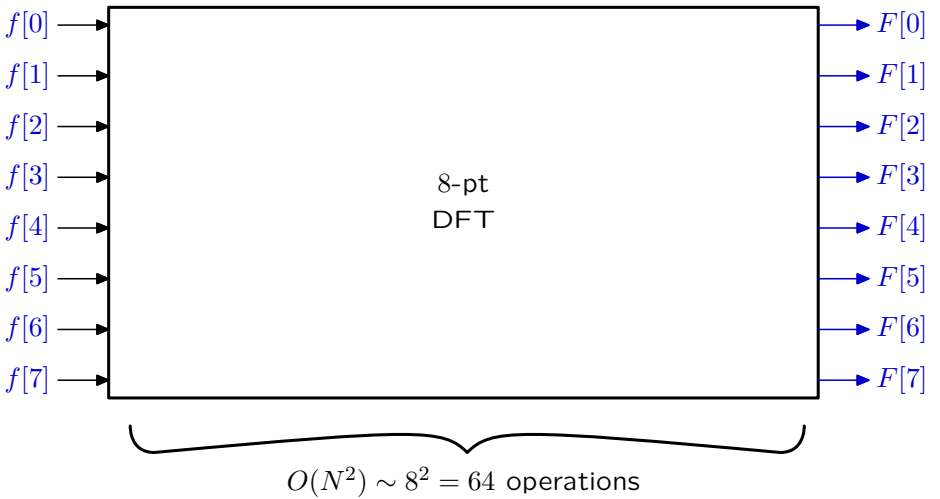
$$
\begin{aligned}
F[k] &= \frac{1}{N}\sum_{n=0}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{\substack{n=0 \\ n \text{ even}}}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} + \frac{1}{N}\sum_{\substack{n=0 \\ n \text{ odd}}}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi k(2m)}{N}} + \frac{1}{N}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi k(2m+1)}{N}} \\
&= \frac{1}{2}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of even numbered inputs}} + \frac{1}{2}e^{-j\frac{2\pi k}{N}}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of odd numbered inputs}}
\end{aligned}
$$

This refactorization reduces an N-point DFT to two $N/2$-point DFTs.

Is that good?

Data Paths

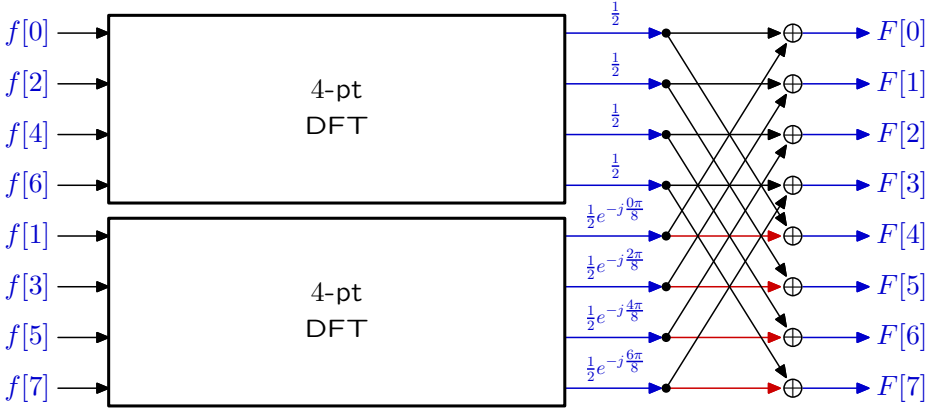
Draw **data paths** to help visualize the FFT algorithm.



Start with an 8-point DFT.

Data Paths

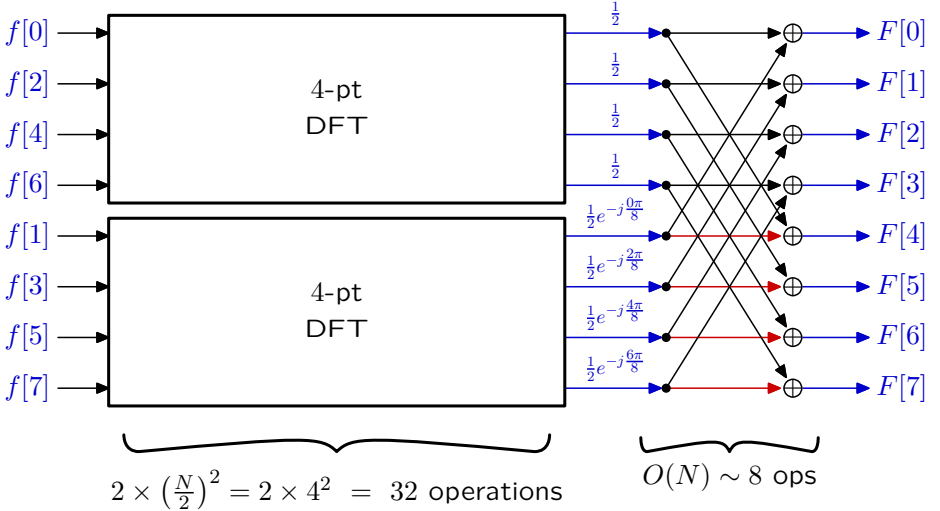
Write the 8-point DFT in terms of the DFTs of even and odd samples.



$$F[k] = \frac{1}{2}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of even numbered inputs}} + \frac{1}{2}e^{-j\frac{2\pi k}{N}}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of odd numbered inputs}}$$

Data Paths

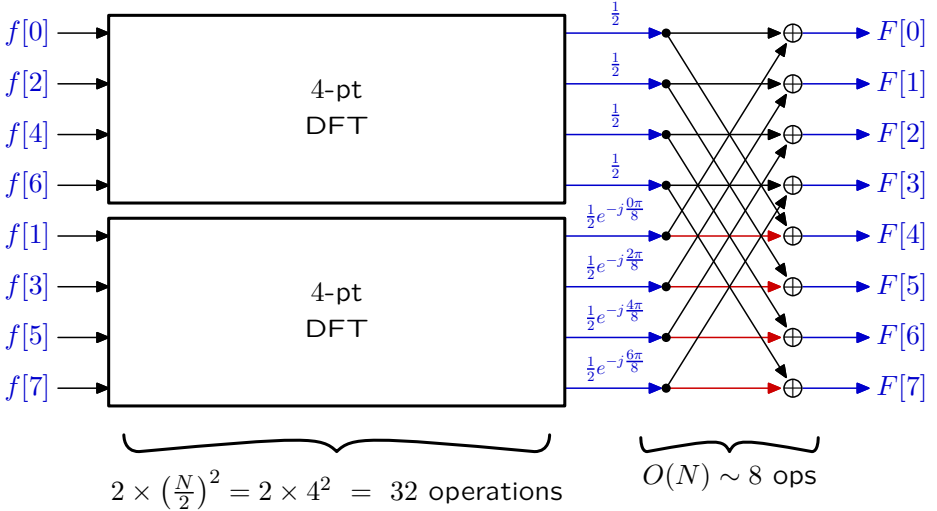
Write the 8-point DFT in terms of the DFTs of even and odd samples.



The numbers above the blue arrows represent multiplicative constants.
The red arrows represent multiplication by $e^{-j\pi} = -1$.

Data Paths

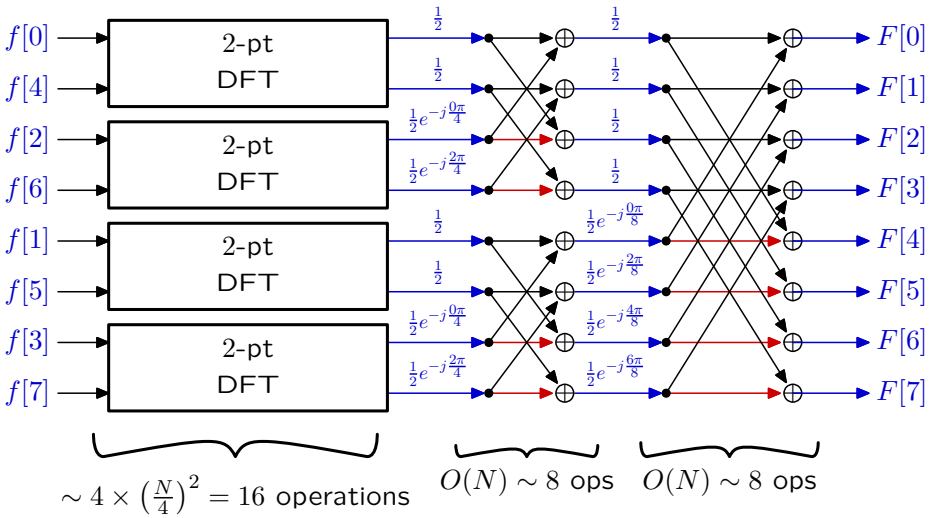
Write the 8-point DFT in terms of the DFTs of even and odd samples.



The number of operations to compute the DFTs is half that of the original. But we have $O(N)$ operations to combine the even and odd results.

Data Paths

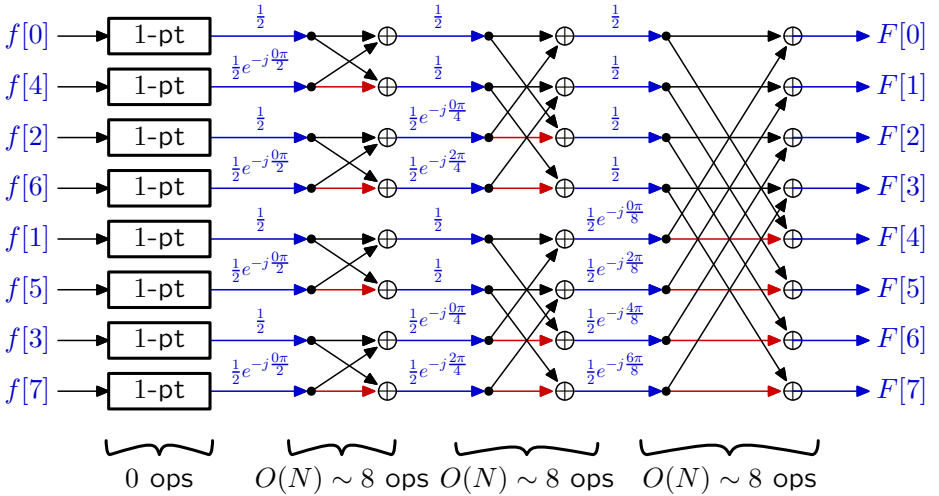
Write the 4-point DFTs in terms of 2-point DFTs.



The number of operations to compute the DFTs is one-fourth that of the original. But we have twice as many operations to combine the parts.

Data Paths

Write the 2-point DFTs in terms of 1-point DFTs.

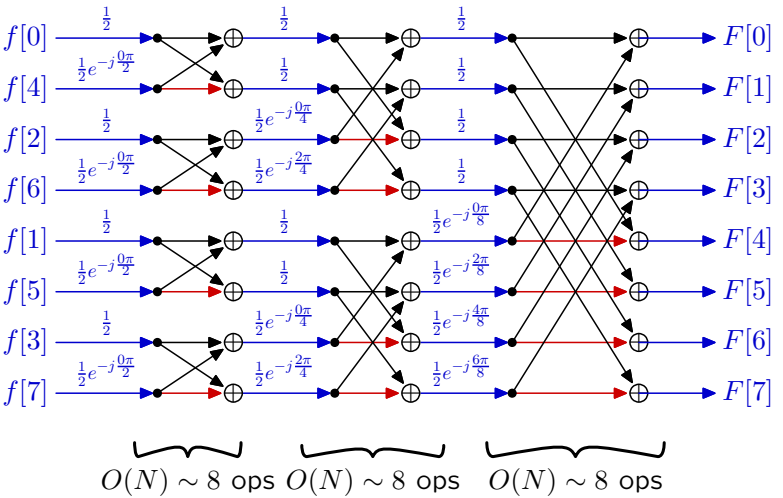


No operations are required to compute the 1-point DFTs.

But we have three times as many operations to combine the parts.

Data Paths

The FFT algorithm reduces the explicit DFTs to length 1.



All that remains to calculate is "glue". There are $\log_2(N)$ stages of glue and each is $O(N)$. So the algorithm is $N \log_2(N)$.

FFT Speedup

The speed of the FFT has had a profound impact on signal processing.

N	DFT	FFT	speed-up	
2	4	2	2.0	
4	16	8	2.0	
8	64	24	2.7	
16	256	64	4.0	
32	1,024	160	6.4	
64	4,096	384	10.7	
128	16,384	896	18.3	
256	65,536	2,048	32.0	
512	262,144	4,608	56.9	
1,024	1,048,576	10,240	102.4	
2,048	4,194,304	22,528	186.2	
4,096	16,777,216	49,152	341.3	
8,192	67,108,864	106,496	630.2	
16,384	268,435,456	229,376	1,170.3	
32,768	1,073,741,824	491,520	2,184.5	
65,536	4,294,967,296	1,048,576	4,096.0	
131,072	17,179,869,184	2,228,224	7,710.1	
262,144	68,719,476,736	4,718,592	14,563.6	
524,288	274,877,906,944	9,961,472	27,594.1	
1,048,576	1,099,511,627,776	20,971,520	52,428.8	
2,097,152	4,398,046,511,104	44,040,192	99,864.4	← 1 minute of audio
4,194,304	17,592,186,044,416	92,274,688	190,650.2	

FFT Speedup

The speed of the FFT has had a profound impact on signal processing **especially multi-dimensional signal processing**!

Consider processing 1080p video images (1920×1080) pixels.

Computing a 2D DFT requires
- one DFT per row + one DFT per column $\sim 2N$ DFTs
- each DFT requires $O(N^2)$ operations
- total is $O(2N \times N^2)$.

Using the FFT reduces this to $O(2N \times N \log_2(N))$: faster by $\approx 175\times$.

FFT reduces times from ≈ 3 hours to about a minute (on my laptop)!

Gauss

Gauss developed the basic idea behind the FFT algorithm in his study of the orbit of the then recently discovered asteroid Pallas. The manuscript was written circa 1805 and published posthumously in 1866.

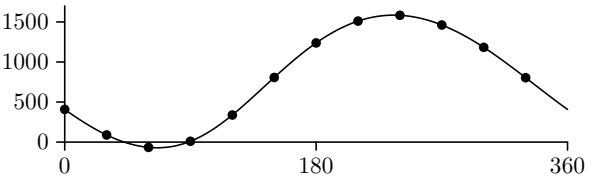
Gauss' data: "declination" X (minutes of arc) v. "ascension" θ (degrees)[4]

θ:	0	30	60	90	120	150	180	210	240	270	300	330
X:	408	89	−66	10	338	807	1238	1511	1583	1462	1183	804

Fitting function:

$$X = f(\theta) = a_0 + \sum_{k=1}^{5} \left[a_k \cos\left(\frac{2\pi k\theta}{360}\right) + b_k \sin\left(\frac{2\pi k\theta}{360}\right) \right] + a_6 \cos\left(\frac{12\pi\theta}{360}\right)$$

Resulting fit:



[4] B. Osgood, "The Fourier Transform and its Applications"

Gauss

Gauss was more interested in understanding the inherent symmetries and using those to generate a robust solution.

Gauss fitted 12 variables to 12 equations.

θ:	0	30	60	90	120	150	180	210	240	270	300	330
X:	408	89	-66	10	338	807	1238	1511	1583	1462	1183	804

Speedup would be $\frac{12\times 12}{12\times \log_2(12)} \approx 3.3$.

Gauss was more interested in understanding than in operation count.

Python Code

Consider the following code to implement the FFT algorithm.

```
from math import e,pi
def FFT(x):
    N = len(x)
    if N==1:
        return x
    if N%2 != 0:
        print('N must be even')
        exit(1)
    xe = x[::2]
    xo = x[1::2]
    Xe = FFT(xe)
    Xo = FFT(xo)
    X = []
    for k in range(N//2):
        X.append((Xe[k]+e**(-2j*pi*k/N)*Xo[k])/2)
    for k in range(N//2):
        X.append((Xe[k]-e**(-2j*pi*k/N)*Xo[k])/2)
    return X
```

Why are there two `for` loops?

Could we substitute a single loop over all N values?

Check Yourself

We can make minor changes to this FFT algorithm to compute the iDFT.

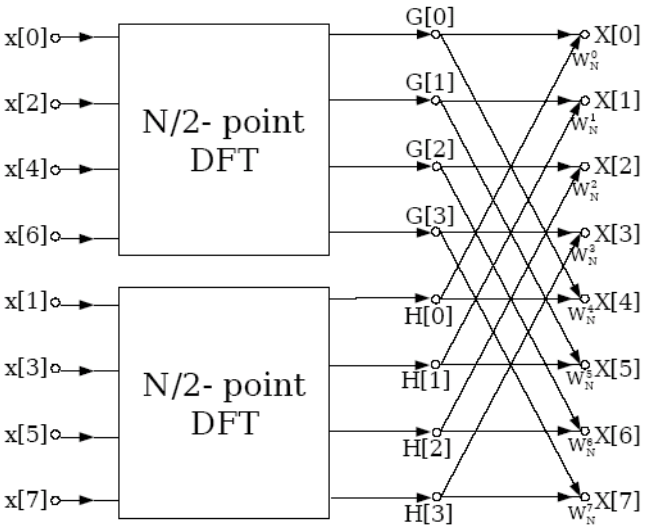
```
from math import e,pi
def FFT(x):
    N = len(x)
    if N==1:
        return x
    if N%2 != 0:
        print('N must be even')
        exit(1)
    xe = x[::2]
    xo = x[1::2]
    Xe = FFT(xe)
    Xo = FFT(xo)
    X = []
    for k in range(N//2):
        X.append((Xe[k]+e**(-2j*pi*k/N)*Xo[k])/2)
    for k in range(N//2):
        X.append((Xe[k]-e**(-2j*pi*k/N)*Xo[k])/2)
    return X
```

Determine the changes that are needed.

Decimation in Time

There are many different "FFT" algorithms.

We have been looking at a "decimation in time" algorithm.

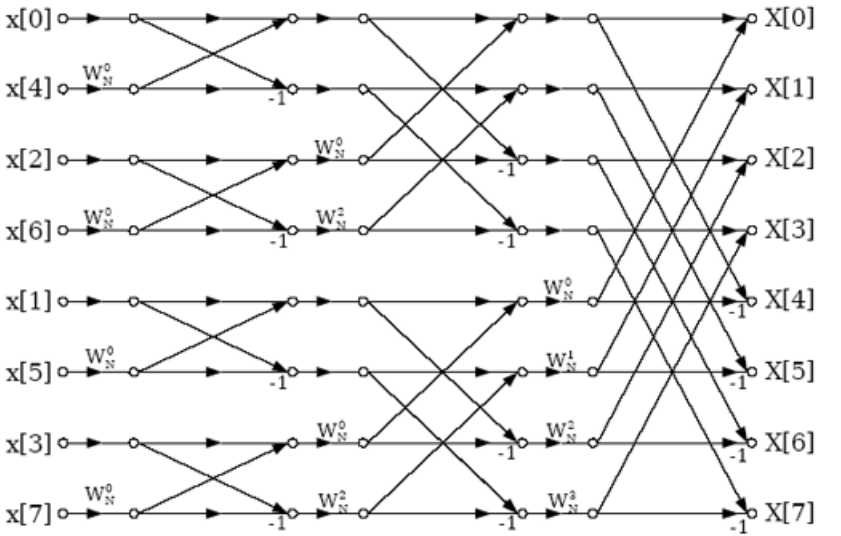


https://cnx.org/contents/qAa9OhlP@2.44:zmcmahhR@7/Decimation-in-time-DIT-Radix-2-FFT

Decimation in Time

There are many different "FFT" algorithms.

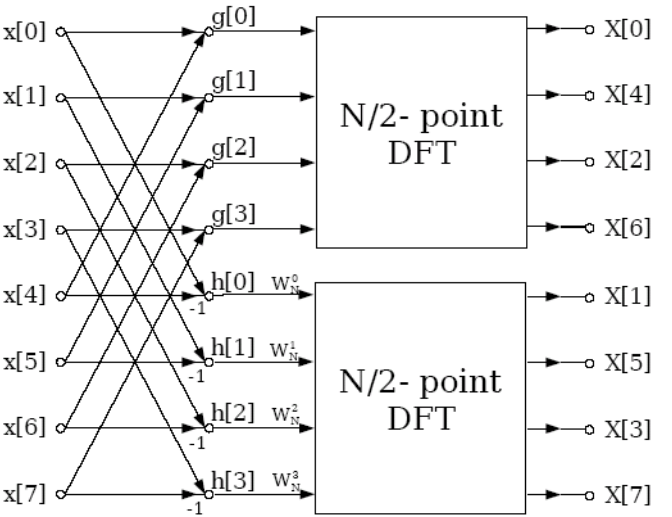
After repeated application of "decimation in time."



Decimation in Frequency

There are many different "FFT" algorithms.

Here is a "decimation in frequency" algorithm.

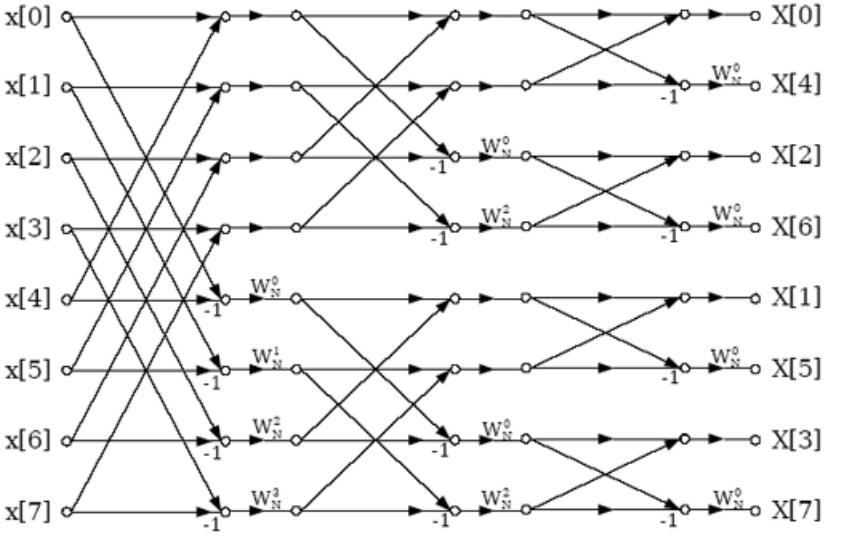


https://cnx.org/contents/qAa9OhlP@2.44:XaYDVUAS@6/Decimation-in-Frequency-DIF-Radix-2-FFT

Decimation in Frequency

There are many different "FFT" algorithms.

After repeated application of "decimation in frequency."



FFTs With Other Radices

What if N is not a power of 2?

Factor N, and apply an algorithm tailored to each factor.

Example: radix 3

$$
\begin{aligned}
F[k] &= \frac{1}{N}\sum_{n=0}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{m=0}^{N/3-1} f[3m]e^{-j\frac{2\pi k(3m)}{N}} + \frac{1}{N}\sum_{m=0}^{N/3-1} f[3m+1]e^{-j\frac{2\pi k(3m+1)}{N}} + \frac{1}{N}\sum_{m=0}^{N/3-1} f[3m+2]e^{-j\frac{2\pi k(3m+2)}{N}} \\
&= \frac{1}{3}\frac{1}{N/3}\sum_{m=0}^{N/3-1} f[3m]e^{-j\frac{2\pi km}{N/3}} \\
&\quad + \frac{1}{3}\frac{1}{N/3}e^{-j2\pi k/N}\sum_{m=0}^{N/3-1} f[3m+1]e^{-j\frac{2\pi km}{N/3}} \\
&\quad + \frac{1}{3}\frac{1}{N/3}e^{-j4\pi k/N}\sum_{m=0}^{N/3-1} f[3m+2]e^{-j\frac{2\pi km}{N/3}} \\
&= \frac{1}{3}\text{DFT(block 0)} + \frac{1}{3}e^{-j\frac{2\pi k}{N}}\text{DFT(block 1)} + \frac{1}{3}e^{-j\frac{4\pi k}{N}}\text{DFT(block 2)}
\end{aligned}
$$

Fast Fourier Transform

The Fast-Fourier Transform (FFT) is an algorithm (actually a family of algorithms) for computing the Discrete Fourier Transform (DFT).

Both elegant and useful, the FFT algorithm is arguably
the most important algorithm in modern signal processing.

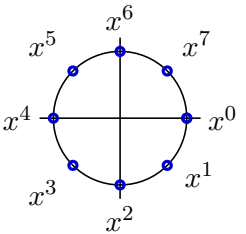
– **computer science** (divide-and-conquer: **elegant** and **efficient**)
– **elegant mathematics** (using alternative representations for polynomials)
– **widely used** in engineering and science (enormous practical value)

The FFT as a Polynomial Representation[5]

Think about the DFT

$$F[k] = \frac{1}{N} \sum_{n=0}^{N-1} f[n] e^{-j\frac{2\pi kn}{N}}$$

as values of an underlying frequency representation $F'(\cdot)$ at points x^k in the complex plane, where $x = e^{-j2\pi/N}$.



$$F[k] = F'(x^k) = \frac{1}{N} \sum_{n=0}^{N-1} f[n](x^k)^n$$

$F'(x^k)$ can be computed as a polynomial in x^k with coefficients $f[n]$. Evaluating the polynomial yields the frequency representation $F'(\cdot)$ and sampling $F'(\cdot)$ at powers of the N^{th} root of unity provides the DFT.

---

[5] https://www.youtube.com/watch?v=h7apO7q16V0 and Prof. Erik Demaine in 6.046 https://www.youtube.com/watch?v=iTMn0Kt18tg