CW1 - CO202 Algorithms 2

Introduction

The Discrete Fourier Transform (DFT) maps tuples $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ of complex numbers to tuples $\mathbf{y} = (y_0, y_1, \ldots, y_{N-1})$ of complex numbers satisfying

$$y_k := \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} = \sum_{n=0}^{N-1} x_n (\cos(2\pi kn/N) + i \sin(2\pi kn/N))$$

for $k = 0, 1, \ldots, N-1$. In the expression above, i denotes the imaginary unit.

We will denote the DFT of $\mathbf{x}$ by $\mathcal{F}(\mathbf{x})$. Then, $\mathcal{F} : \mathbb{C}^N \to \mathbb{C}^N$ is a linear, invertible transformation, where $\mathbb{C}$ denotes the set of complex numbers.

We can recover $\mathbf{x}$ from $\mathbf{y} = \mathcal{F}(\mathbf{x})$ by computing

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} y_n e^{\frac{2\pi i}{N} kn}$$

for $k = 0, 1, \ldots, N-1$. This formula leads to a simple way of computing $\mathbf{x}$ from $\mathbf{y} = \mathcal{F}(\mathbf{x})$ that only requires an algorithm for the forward DFT $\mathcal{F}$. More precisely, we have

$$x_k = \frac{\text{swap}\left(\mathcal{F}(\text{swap}(\mathbf{y}))_k\right)}{N}$$

for $k = 0, 1, \ldots, N-1$, where $\text{swap}(a + bi) = b + ai$.

The DFT has many practical applications in computer science and electrical engineering. It is widely used in signal and image processing, and it even makes a crucial appearance in the fastest known algorithm for multiplying large integers. Because of this, it is of the utmost interest to find efficient algorithms for computing the DFT.

Currently, the most efficient algorithm for computing the DFT is called the Fast Fourier Transform (FFT). It has time complexity $O(N \log N)$ (if one assumes arithmetic operations take time $O(1)$), and is an important example of the divide-and-conquer approach to algorithm design. As a consequence, by the discussion above, there is also an algorithm computing the inverse DFT in time $O(N \log N)$.

In [3]:
```
#You will need these packages to complete the coursework

import numpy as np
import math
import cmath
```

Exercise 1 (warmup)

Assuming arithmetic operations take time $O(1)$, write a Python program for computing $\mathcal{F}(\mathbf{x})$ on input $\mathbf{x} \in \mathbb{C}^N$ in time $O(N^2)$. Briefly argue that the asymptotic time complexity of the proposed algorithm is indeed $O(N^2)$.

Answer

For each $k = 0, 1, \ldots, N-1$, compute each term in the sum defining y_k separately. Since there are N sums and N terms per sum, the time complexity of this algorithm is $O(N^2)$.

In [24]:
```
def naiveFFT(x):

    y = np.zeros(len(x),dtype=complex)

    for k in range(len(x)):

        for n in range(len(x)):

            y[k] = y[k] + x[n] * cmath.exp(-2 * math.pi * 1j * k * n / len(x))

    return y
```

Exercise 2 (FFT)

In this exercise, you will implement the Cooley-Tukey FFT algorithm, which computes $\mathcal{F}(\mathbf{x})$ on input $\mathbf{x} \in \mathbb{C}^N$ in time $O(N \log N)$ when N is a power of 2.

Suppose that N is a power of 2, i.e., we have $N = 2^r$ for some non-negative integer r. For $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$, define

$$\mathbf{x}_{\text{odd}} = (x_1, x_3, x_5, \ldots, x_{N-1}) \in \mathbb{C}^{N/2},$$

and

$$\mathbf{x}_{\text{even}} = (x_0, x_2, x_4, \ldots, x_{N-2}) \in \mathbb{C}^{N/2}.$$

Let $\mathbf{y}$, $\mathbf{y}_{\text{odd}}$, and $\mathbf{y}_{\text{even}}$ be the DFT's of $\mathbf{x}$, $\mathbf{x}_{\text{odd}}$, and $\mathbf{x}_{\text{even}}$, respectively. From the properties of $\mathcal{F}$, we can derive the identities

$$\mathbf{y}_k = (\mathbf{y}_{\text{even}})_k + e^{-\frac{2\pi i}{N}k}(\mathbf{y}_{\text{odd}})_k,$$

and

$$\mathbf{y}_{k+N/2} = (\mathbf{y}_{\text{even}})_k - e^{-\frac{2\pi i}{N}k}(\mathbf{y}_{\text{odd}})_k$$

for $k = 0, 1, \ldots, N/2 - 1$.

Using a divide-and-conquer approach and the two identities above, write a Python program below that computes $\mathbf{y} = \mathcal{F}(\mathbf{x})$ in time $O(N \log N)$. You may assume that the length N of the input $\mathbf{x}$ is always a power of 2.

Answer

In [50]:
```
def FFT(x):

    if len(x)==1:

        #If x is a list with a single element, then its DFT is x
        return x

    else:

        #Split x into even- and odd-numbered entries
        xEven=x[0::2]
        xOdd=x[1::2]

        #Compute FFT of xEven, xOdd
        yEven=FFT(xEven)
        yOdd=FFT(xOdd)

        #Initialize output list
        y=np.zeros(len(x),dtype=complex)

        #Use two identities above to compute y from yEven and yOdd
        for k in range(len(x)//2):

                y[k] = yEven[k] + cmath.exp(-2 * math.pi * 1j * k /
len(x)) * yOdd[k]

                y[k+len(x)//2] = yEven[k] - cmath.exp(-2 * math.pi
* 1j * k / len(x)) * yOdd[k]

        return y
```

In [89]:
```
%timeit -n 10 FFT(np.ones(1000))
```

```
10 loops, best of 3: 16.7 ms per loop
```

In [90]:
```
%timeit -n 10 naiveFFT(np.ones(1000))
```

```
10 loops, best of 3: 2.56 s per loop
```

Exercise 3 (runtime comparison)

In this exercise, you will compare the runtime of the two algorithms you implemented in Exercises 1 and 2.

Run the algorithms you implemented in Exercises 1 and 2 and plot their runtimes on the inputs below

np.ones(10)

np.ones(30)

np.ones(50)

np.ones(100)

np.ones(200)

np.ones(500)

np.ones(1000)

In [93]:
```
#To measure runtime in a reliable way you can use the timeit functi
on.

# The call below runs 20 loops, and in each loop runs FFT(np.ones(1
00)) 3 times and takes the best time.
# The final output is the average of the 20 loops.
%timeit -n 20 FFT(np.ones(1000))
```

```
20 loops, best of 3: 15.6 ms per loop
```

Exercise 4 (application)

In this exercise, you will implement your very own efficient image compression method based on the DFT from scratch.

The 2-dimensional DFT of an $N_1 \times N_2$ matrix $X = (X_{n_1,n_2})$ is an $N_1 \times N_2$ matrix $Y = (Y_{k_1,k_2})$ satisfying

$$Y_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} e^{-2\pi i \left(\frac{k_1 n_1}{N_1} + \frac{k_2 n_2}{N_2} \right)} X_{n_1,n_2}$$

for every $k_1 = 0, 1, \ldots, N_1 - 1$ and $k_2 = 0, 1, \ldots, N_2 - 1$. Similarly to the 1-dimensional case, we may also write $Y = \mathcal{F}(X)$.

- ## Exercise 4.1 The 2-dimensional DFT $\mathcal{F}(X)$ can be easily computed using only the 1-dimensional DFT via the so-called row-column algorithm, which we describe below. For a matrix M, let $M_{i,\cdot}$ denote the i-th row of M, and let $M_{\cdot,j}$ denote the j-th column of M.

- On input an $N_1 \times N_2$ matrix X, compute $\mathcal{F}(X_{i,\cdot})$ for $i = 0, 1, \ldots, N_1 - 1$. Let Z be the $N_1 \times N_2$ matrix such that $Z_{i,\cdot} = \mathcal{F}(X_{i,\cdot})$.
- For each $j = 0, 1, \ldots, N_2 - 1$, compute $\mathcal{F}(Z_{\cdot,j})$. Let Y be the $N_1 \times N_2$ matrix such that $Y_{\cdot,j} = \mathcal{F}(Z_{\cdot,j})$. Then, we have $Y = \mathcal{F}(X)$.

Assuming N_1 and N_2 are powers of 2, implement the row-column algorithm for computing the 2-dimensional DFT of X using the FFT algorithm from Exercise 2.

- ## Exercise 4.2 Recall from the Introduction that, in the 1-dimensional case with $\mathbf{x}, \mathbf{y} \in \mathbb{C}^N$ and $\mathcal{F}(\mathbf{x}) = \mathbf{y}$, we have

$$x_k = \frac{\text{swap}\,(\mathcal{F}(\text{swap}(\mathbf{y}))_k)}{N}$$

for $k = 0, 1, \ldots, N - 1$, where $\text{swap}(a + bi) = b + ai$.

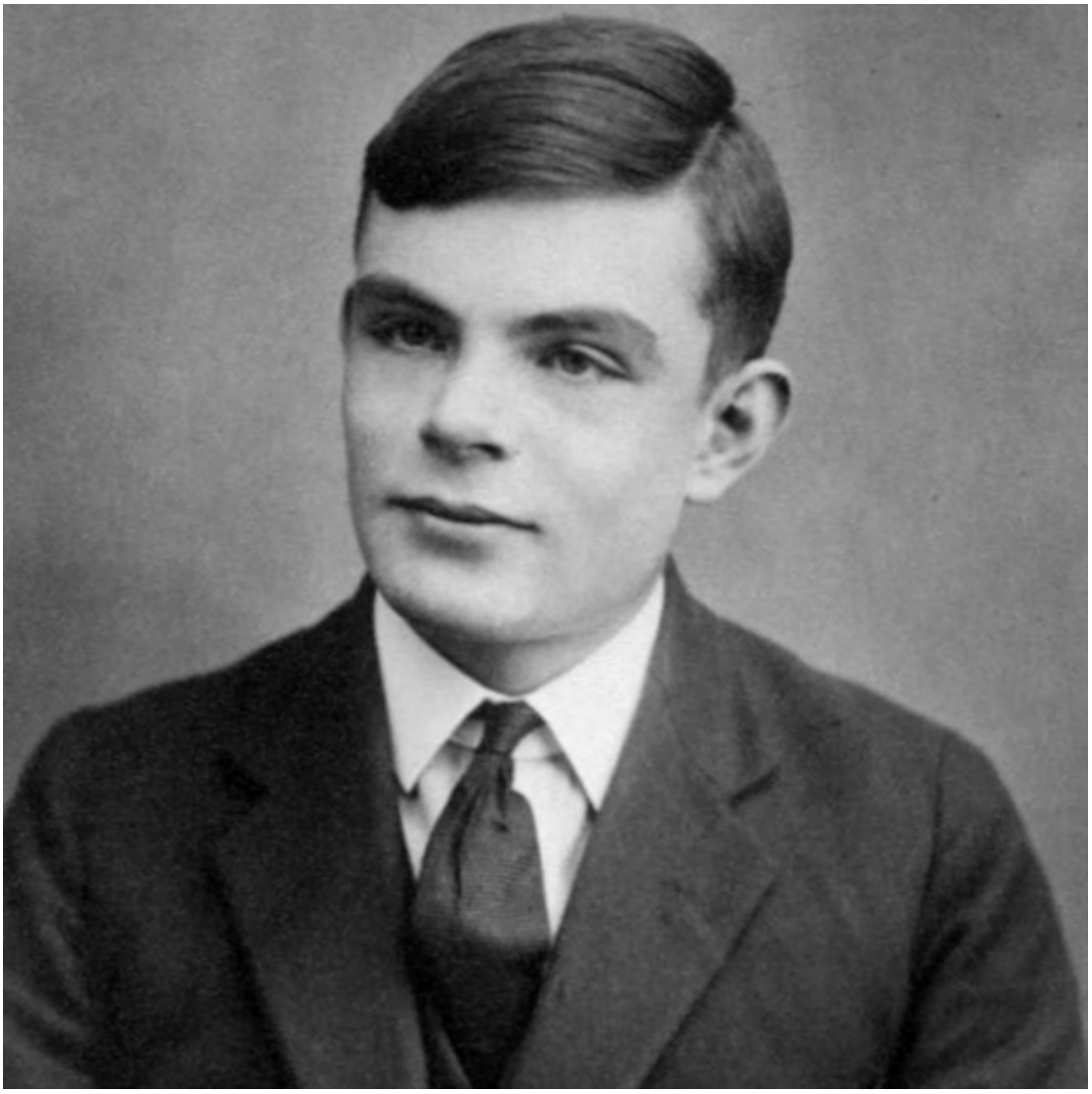
Combining this observation with the row-column algorithm described in Exercise 4.1, implement an algorithm that recovers $X \in \mathbb{C}^{N_1 \times N_2}$ from $Y = \mathcal{F}(X)$ in time $O(N_1 N_2 \log(N_1 N_2))$ whenever N_1 and N_2 are powers of 2. Argue why the running time of your algorithm is as required.

- ## Exercise 4.3 You are now all set to implement the DFT-based image compression method. This method receives as input a matrix X (which encodes a grayscale image) and a threshold $t > 0$, and works as follows:
  - Compute $Y = \mathcal{F}(X)$ using the algorithm from Exercise 4.1.
  - Compute $\tilde{Y}$ from Y by setting $\tilde{Y}_{k_1,k_2} = 0$ if $|Y_{k_1,k_2}| < t$, and $\tilde{Y}_{k_1,k_2} = Y_{k_1,k_2}$ otherwise. Here, $|z| = \sqrt{a^2 + b^2}$ is the modulus of a complex number $z = a + bi$.
  - Invert $\tilde{Y}$ using the algorithm from Exercise 4.2 to obtain $\tilde{X}$, which encodes the compressed image.

Implement this algorithm.

- ## Exercise 4.4

To conclude, you will compress the image below



Apply your DFT-based compression method to the image above. Experiment with different thresholds, and showcase some of your attempts! You will find some code below for converting a grayscale image into an array, and for converting the output of the compression algorithm into a grayscale image.

In [ ]:
```
#After saving the image above to your computer, you can convert it
to matrix form using the following code.

from PIL import Image

img = Image.open('path to your image').convert('L')

img_array = np.array(img)

img_array_complex = img_array.astype(complex)

#You should apply your compression algorithm to img_array_complex
```

In [ ]:
```
#You can convert the matrices you obtain into grayscale images with
the following code

#compressed_img_array is the output of your compression algorithm

#discard the imaginary part and round the values in your array to t
he nearest integer
compressed_img_array_int = compressed_img_array.astype(np.uint8)

#convert the rounded array into a grayscale image
compressed_img = Image.fromarray(compressed_img_array_int, 'L')

#save the image
compressed_img.save('/Users/joaoribeiro/Downloads/compressed-turing
.png')
```

Answer 4.1

In [40]:
```
# x is numpy array
def FFT2(x):

    #compute FFT of each row of x
    z = np.zeros((len(x),len(x[0])),dtype=complex)

    for i in range(len(x)):

         z[i]=FFT(x[i])

    #compute FFT of each column of z
    y = np.zeros((len(x),len(x[0])),dtype=complex)

    for j in range(len(x[0])):

        y[:,j] = FFT(z[:,j])

    return y
```

Answer 4.2

In [78]:
```
# swap operation
def swap(z):
    return z.imag + z.real*1j

# 1d inverse FFT
def invFFT1(y):

    #swap entries of y
    yswap = np.zeros(len(y),dtype=complex)

    for k in range(len(y)):
        yswap[k]=swap(y[k])

    #compute FFT of swapped y
    xtemp = FFT(yswap)

    #for final output, simply swap elements of xtemp and divide by
N
    x = np.zeros(len(y),dtype=complex)

    for k in range(len(y)):

        x[k] = swap(xtemp[k])/len(y)

    return x

# 2d inverse FFT
def invFFT2(y):

    #invert the columns of y to obtain z
    z = np.zeros((len(y),len(y[0])),dtype=complex)

    for j in range(len(y[0])):

        z[:,j] = invFFT1(y[:,j])

    #invert the rows of z to obtain x
    x = np.zeros((len(y),len(y[0])),dtype=complex)

    for i in range(len(y)):

        x[i] = invFFT1(z[i])

    return x
```

Answer 4.3

In [156]:
```
# x encodes a grayscale image as numpy array

def compressFFT(x,t):

    #compute 2d FFT of x
    y = FFT2(x)


    #remove small Fourier coefficients
    for i in range(len(x)):

        for j in range(len(x[0])):

            if abs(y[i,j]) < t:

                y[i,j] = 0

    #invert modified y to obtain compressed x
    xcompressed = invFFT2(y)

    return xcompressed
```

Answer 4.4

In [208]:
```
from PIL import Image

img = Image.open('/Users/joaoribeiro/Downloads/turing2.png').conver
t('L')
img_array = np.array(img)

img_array_complex = img_array.astype(complex)

compressed_img_array = compressFFT(img_array_complex,35000)

compressed_img_array_int = compressed_img_array.astype(np.uint8)

compressed_img_array_int

compressed_img = Image.fromarray(compressed_img_array_int, 'L')

compressed_img.save('/Users/joaoribeiro/Downloads/compressed-turing
.png')
```

```
/usr/local/lib/python2.7/site-packages/ipykernel_launcher.py:10: C
omplexWarning: Casting complex values to real discards the imagina
ry part
  # Remove the CWD from sys.path while we load stuff.
```