Examinee number: 11070
Name: Gábor Ádám Fehér

Answer to Theme A

From the fundamental concepts related to mathematical informatics we present the **Cooley–Tukey algorithm**. The algorithm allows us to compute the discrete Fourier transform of a n long sequence in $\mathcal{O}(n \log n)$ time. Algorithm with such properties are called fast Fourier transform (FFT) algorithms. Among the many variants of the Cooley–Tukey algorithm, the **radix-2 DIT** is the simplest and most common one, and thus we present it in great detail. Other forms of the algorithm, as well as other types of FFT algorithms, are also mentioned but no rigorous construction is given.

Mathematical overview

Definition 1 (Discrete Fourier transform)**.** The discrete Fourier transform (DFT) over the n dimensional complex field is an invertible linear transformation $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^n$. The output of the function for $(x_k)_{0 \le k \le n-1}$ is defined by

$$X_k = (\mathcal{F}(x))_k = \sum_{j=0}^{n-1} x_j e^{-\frac{2\pi i k j}{n}} = \sum_{j=0}^{n-1} x_j \left[\cos\left(\frac{2\pi i k j}{n}\right) - i\sin\left(\frac{2\pi i k j}{n}\right) \right], \tag{1}$$

where the right side of the equation is due to Euler's formula and the trigonometric properties of sine and cosine.

We should mention that the sign of the exponential is sometimes taken as positive. The resulting equation is equal to the inverse of the previously defined Fourier transformation multiplied by the constant n, that is

$$x_k = \left(\mathcal{F}^{-1}(X)\right)_k = \frac{1}{n} \sum_{j=0}^{n-1} X_j e^{\frac{2\pi i k j}{n}}. \tag{2}$$

With some minor adjustments the Cooley–Tukey algorithm is capable of computing the inverse-DFT of the n long sequence, so no matter which convention we use, the algorithm remains useful.

We introduce some notations. Given $x = (x_k)_{0 \le 2n-1}$, we define $X^{[0]}$ to be the DFT of the even-indexed terms, while $X^{[1]}$ to be the DFT of the odd-indexed terms in the sequence. That is

$$X_k^{[0]} = \left(X^{[0]}\right)_k = (\mathcal{F}(x_0, x_2, \ldots x_{2n-2}))_k = \sum_{j=0}^{n-1} x_{(2j)} e^{-\frac{2\pi i k j}{n}} = \sum_{j=0}^{n-1} x_{(2j)} e^{-\frac{4\pi i k j}{2n}} \tag{3}$$

$$X_k^{[1]} = \left(X^{[1]}\right)_k = (\mathcal{F}(x_1, x_3, \ldots x_{2n-1}))_k = \sum_{j=0}^{n-1} x_{(2j+1)} e^{-\frac{2\pi i k j}{n}} = \sum_{j=0}^{n-1} x_{(2j+1)} e^{-\frac{4\pi i k j}{2n}}. \tag{4}$$

The key observation to make in order to verify the validity of the algorithm is

$$X_k = \sum_{j=0}^{2n-1} x_j e^{-\frac{2\pi i k j}{2n}} = \sum_{j=0}^{n-1} x_{(2j)} e^{-\frac{4\pi i k j}{2n}} + e^{-\frac{2\pi i k}{2n}} \sum_{j=0}^{n-1} x_{(2j+1)} e^{-\frac{4\pi i k j}{2n}}. \tag{5}$$

Thus for $0 \le l \le n-1$ we have

$$X_l = X_l^{[0]} + e^{-\frac{2\pi i k}{2n}} X_l^{[1]}, \text{ and } X_{n+l} = X_l^{[0]} - e^{-\frac{2\pi i k}{2n}} X_l^{[1]}, \tag{6}$$

since

$$\sum_{j=0}^{n-1} x_{(2j)} e^{-\frac{4\pi i (n+l) j}{2n}} = \sum_{j=0}^{n-1} x_{(2j)} e^{-\frac{4\pi i l j}{2n}}, \text{ and } e^{-\frac{2\pi i (n+l)}{2n}} = -e^{-\frac{2\pi i l}{2n}}. \tag{7}$$

The algorithm

This basis of all variants of the Cooley–Tukey algorithm is the divide-and-conquer technique. In the radix-2 case the size of the input has to be a power of two. We may add padding zeros to fit the size constraint. From this point on we assume that input vectors are of the right size. Given an $x_{0 \le k \le n}$, where $2 \le n$, equations 3 and 4 indicate that the input vector should be divided into two parts: the even-indexed and the odd-indexed terms. Once the DFTs of the split vector are calculated, via recursive calls, the DFT of the input vector can be calculated by using equation 6. If the input vector is one dimensional, its DFT is itself. The following Python code is a working implementation of the radix-2 case, illustrating the key ideas used, but due to its performance it is not meant to be used in real world applications.

```python
1    """A working fft implementation"""
2    from sympy import exp, pi, I
3
4    def fft(terms):
5        """Implementation of Cooley-Tukey FFT algorithm"""
6        input_length = len(terms)
7        if input_length == 1:
8            return terms
9        minus_nth_root_of_unity = exp(-2*pi*I/input_length)
10       running_root = 1
11       even_indexed_terms = [terms[i] for i in range(0, input_length, 2)]
12       odd_indexed_terms = [terms[i] for i in range(1, input_length, 2)]
13       even_fft = fft(even_indexed_terms)
14       odd_fft = fft(odd_indexed_terms)
15       left_fft, right_fft = [], []
16       for k in range(0, int(input_length/2)):
17           sub_expr = running_root * odd_fft[k]
18           left_fft.append(even_fft[k] + sub_expr)
19           right_fft.append(even_fft[k] - sub_expr)
20           running_root *= minus_nth_root_of_unity
21       return left_fft + right_fft
```

To evaluate the time complexity of the algorithm, apart from the recursive calls in line 13 and 14, the algorithm runs in $\Theta(n)$ time, where n is the length of the input. Using the recurrence relation

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n). \tag{8}$$

With little modification, the Cooley–Tukey algorithm can compute the inverse-DFT of a transformed vector. Similarly to equation 5, we can derive

$$x_k = \sum_{j=0}^{2n-1} X_j e^{\frac{2\pi i k j}{2n}} = \sum_{j=0}^{n-1} X_{(2j)} e^{\frac{4\pi i k j}{2n}} + e^{\frac{2\pi i k}{2n}} \sum_{j=0}^{n-1} X_{(2j+1)} e^{\frac{4\pi i k j}{2n}}, \tag{9}$$

so in the 9th line, instead of $e^{-2\pi i/N}$ we have $e^{2\pi i/N}$ for the N long input, and by multiplying each term of the end result by $1/n$ given the initial input had size of n, we obtain the original x, within the same time complexity.

Importance and Usage

The FFT is widely used among different fields of engineering, computer science and mathematics.

Many popular compression methods, such as `jpeg` or `webm`, use discrete cosine transforms (DCT). Algorithms that compute DCTs in $\mathcal{O}(n \log n)$ are known as fast cosine transform (FCT) algorithms. Specialized FCT algorithms exist, that directly compute a vectors DCT, and tough the theoretical running time of such algorithms are better, on modern hardware computing DCTs via FFTs with some $\mathcal{O}(n)$ pre- and post-processing steps are often faster.

When two degree-bound n polynomials are represented in the a "extended" point-value form, over $2n$ distinct points, multiplication can be done in $\mathcal{O}(n)$ time, by simply multiplying the coefficients. Representing such a polynomial over the nth root of unity is the same as having the coefficient vectors transformed. Thus multiplying two polynomials can be done in $\mathcal{O}(n \log n)$ time by using FFTs: first transforming the polynomials to their "extended" point value form in $\mathcal{O}(n \log n)$ time, multiplying them in $\mathcal{O}(n)$, and then transforming back the result to coefficient form. Polynomial representation of the Toeplitz matrices also allow us to multiply them within the same time complexity.

FFts are also widely used in theoretical results. For a long time the **Schönhage–Strassen algorithm** was the fastest known algorithm for multiplying two large integers. With running time $\mathcal{O}(n \log n \log \log n)$, it is asymptotically faster than other many traditional multiplication methods, like for example the **Karatsuba algorithm**, but it only starts to outperform them for numbers with 10000 to 40000 digits, thus it is rarely used in practice. The algorithm employs modular arithmetic instead of the roots of unity (Fourier transforms can be performed in any algebric ring). In 2019 Harvey and van der Hoeven published a $\mathcal{O}(n \log n)$ algorithm, which also builds upon on FFTs. However this is a galactic algorithm, so it only bears theoretical importance.

Answer to Theme B

Answer to Theme C