ALGORITHMS FINAL PROJECT

Bartoli Ludovico

29 giugno 2021

1 Introduction

The aim of this project is to reproduce the Fast Fourier Transform (FFT) algorithm through a series of gradual improvements in efficiency, both in space and time. This is a great example of how the development of fast algorithms usually consists of using special properties of the algorithm of interest with the simultaneous goals to divide the problem in subproblems and remove redundant operations of a direct implementation.

We will call FFT any algorithm which computes the Discrete Fourier Transform (DFT) of a sequence of N complex numbers (we will consider only the case when N is a power of 2, which is said to be "radix 2") which takes a computational time of O(Nlog(N)). From now on "DFT" indicates both the Discrete Fourier Transform and any algorithm which computes it in a computational time of $O(N^2)$.

Eventually we will present an application for fast multiplication of polynomials, comparing the FFT method with another implementation of polynomials using linked lists.

2 Historical Notes and Applications

The history of the Fast Fourier Transform (FFT) starts in 1805, when Carl Friedrich Gauss tried to interpolate the orbit of certain asteroids from sample locations. Thereby he developed the Discrete Fourier Transform , even before Fourier published his results in 1822. To calculate the DFT he invented an algorithm which is equivalent to the one that Cooley and Tukey published in 1965, describing how to implement it conveniently on a computer (the radix 2 algorithm is only a special case of the more general algorithm developed by Cooley and Tukey). However, Gauss never published his approach or algorithm in his lifetime. It appeared that other methods seemed to be more useful to solve this problem. Probably, that is why nobody realized this manuscript when Gauss' collected works were published in 1866. It took another 160 years until Cooley and Tukey reinvented the FFT. In that time the US-military was interested in a method to detect Soviet nuclear tests. One approach was to analyze seismological time-series data and Tukey was in President Kennedy's Science Advisory Committee that handled the problem.

Popularity of the FFT is evidenced by the wide variety of application areas. In addition to conventional radar, communications, sonar, and speech signal-processing applications, current fields of FFT usage include biomedical engineering, imaging (for example it is fundamental for image compression), analysis of stock market data, spectroscopy, metallurgical analysis, nonlinear

systems analysis, PDEs, mechanical analysis, geophysical analysis, simulation, music synthesis, and the determination of weight variation in the production of paper from pulp. The increase in computer speed by the FFT has revolutionized the use of the discrete Fourier transform in all these fields.

3 Definition of the problem

Definition 3.1. *Given N complex numbers* $\{h_j\}_{j=0}^{N-1}$*, their N-point Discrete Fourier Transform (DFT) is denoted by* $\{H_k\}_{j=0}^{N-1}$ *where* H_k *is defined by*

$$H_k = \sum_{j=0}^{N-1} h_j e^{-i2\pi jk/N} := \sum_{j=0}^{N-1} h_j W^{kj}$$

where $k = 0, 1, ..., N-1$ *and* $W = e^{-i2\pi/N}$ *is one of the N-th complex roots of unity (note that all the other roots are powers of W).*

These computations can be written as a matrix-vector-multiplication $H = W_N h$ *with* $(W_N)_{(k+1,j+1)} = W^{kj} \in \mathbb{R}^{NxN}$ *where* $k, j = 0, ..., N-1$*:*

$$\begin{bmatrix} H_0 \\ H_1 \\ \vdots \\ H_{N-1} \end{bmatrix} = \begin{bmatrix} (W^{0j} = 1)_{j=0,...,N-1}^T \\ \\ (W^{kj})_{j=0,...,N-1}^T \\ \\ (W^{(N-1)j})_{j=0,...,N-1}^T \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{N-1} \end{bmatrix}$$

It is clear that, once we have computed the matrix W_N (of which we know only the first row and first column of ones) and stored its $(N-1)^2$ unknown elements, in the trivial case, upon which we will built the DFT algorithm, we have to perform $\Theta(N^2)$ complex additions and multiplications to obtain the result. First of all we will examine an easier way to perform the multiplications ZW^m, where Z is any complex number, without storing all the elements of W_N and using less multiplications.

4 Rotations

Suppose now $N = 2^R$; we want to compute all the complex products $h_j W^m$ where $h_j := A + iB = Z$ is any complex number and m=kj; therefore $m \in \{0, ..., (N-1)^2\}$. Since $W^m = cos(\frac{2\pi m}{N}) - isin(\frac{2\pi m}{N})$ the multiplication can be interpreted as a rotation of the vector (A,B) through the angle $-2\pi m\mathbb{N}$ in the complex plane. First of all note that, since $W^N = 1$ the function $m \to W^m$ is N-periodic and we can suppose $m < N$ up to substitution with

its remainder equivalent module N. Suppose now $m < N/4$ and that we have already computed all the values

$$S(m) := sin(2\frac{\pi m}{N}) \quad T(m) := tan(\frac{\pi m}{N}).$$

Using algebra and the properties of trigonometric functions (see [1]) we obtain that:

$$\begin{aligned} V &:= T(m)A - B \\ Re(W^m Z) &= A - S(m)V \\ Im(W^m Z) &= -V - T(m)Re(W^m Z) \end{aligned}$$

this three step process requires only 3 real multiplications and 3 additions against the 4 real multiplications and 2 real additions required for standard complex multiplications. This represents a saving when a computer is used that takes significantly longer to perform multiplications than additions. But this is not the only saving: this reduction saves memory storage, since we will show now we need only to store the $2\frac{N}{4}$ sines and tangents to perform all the multiplications against the initial $(N-1)^2$ unknown elements of the matrix W_N. Indeed

- if $m \in \{\frac{N}{4}, ..., \frac{N}{2} - 1\}$, since $W^{\frac{N}{4}} = -i$

  $$W^m Z = W^{m-\frac{N}{4}}(B - iA)$$

  By setting $m' = m - \frac{N}{4}$ and using the previous formula we get

  $$\begin{aligned} V &:= T(m')B + A \\ Re(W^m Z) &= B - S(m')V \\ Im(W^m Z) &= -V - T(m')Re(W^m Z) \end{aligned}$$

- if $m \in \{\frac{N}{2}, ..., \frac{3N}{4} - 1\}$, since $W^{\frac{N}{2}} = -1$

  $$W^m Z = -W^{m-\frac{N}{2}} Z$$

  since $m' = m - \frac{N}{2} < m - \frac{N}{4}$ we obtain

  $$\begin{aligned} Re(W^m Z) &= -Re(W^{m'} Z) \\ Im(W^m Z) &= -Im(W^{m'} Z) \end{aligned}$$

- if $m \in \{\frac{3N}{4}, ..., N-1\}$

$$W^m Z = -W^{m-\frac{N}{2}} Z$$

since $m' = m - \frac{N}{2} \in \{\frac{N}{4}, ..., \frac{N}{2} - 1\}$ we obtain

$$Re(W^m Z) = -Re(W^{m'} Z)$$
$$Im(W^m Z) = -Im(W^{m'} Z)$$

Therefore we proved that only the values $\{T(m)\}_{m=0}^{N/4-1}$ and $\{S(m)\}_{m=0}^{N/4-1}$ are needed to compute the multiplications. The procedure ROTATION takes as input a complex number Z, $N = 2^R$, the exponent m and returns a complex number result of the multiplication ZW^m. In our purposes we are also assuming we have already stored the necessary values of sines and tangents in two arrays S and T, which we will consider input variables as well. ROTATION takes $\Theta(1)$ primitive operations.

Algorithm 1 ROTATION(Z, N, m, T, S)

Require: $N \geq 4, \ \ m \geq 0$

1: $A \leftarrow Z.real$
2: $B \leftarrow Z.imag$
3: $m \leftarrow m\%N$
4: **if** $m < N/4$ **then**
5: $\quad V \leftarrow T[m] * A - B$
6: $\quad Re \leftarrow A - S[m] * V$
7: $\quad Im \leftarrow -V - T[m] * Re$
8: **else if** $m < N/2$ **then**
9: $\quad V \leftarrow T[m - N/4] * B + A$
10: $\quad Re \leftarrow B - S[im - N/4] * V$
11: $\quad Im \leftarrow -V - T[m - N/4] * Re$
12: **else if** m<3N/4 **then**
13: $\quad V \leftarrow T[int(m - N/2)] * A - B$
14: $\quad Re \leftarrow -(A - S[m - N/2] * V)$
15: $\quad Im \leftarrow -(-V - T[m - N/2] * (-Re))$
16: **else**
17: $\quad V \leftarrow T[m - 3 * N/4] * B + A$
18: $\quad Re \leftarrow -(B - S[m - 3 * N/4] * V)$
19: $\quad Im \leftarrow -(-V - T[m - 3 * N/4] * (-Re))$
20: **end if**
21: **return** Re+i Im

5 DFT Algorithm and Analysis

Since we want to apply the procedure ROTATION in our implementation of the DFT algorithm, we will assume that N is a power of two.

Algorithm 2 DFT(h)

Require: h list of complex numbers
1: $N \leftarrow h.length$
2: Let H be a list on length N
3: **if** N=2 **then**
4: $H_0 = h_0 + h_1$
5: $H_1 = h_0 - h_1$
6: **return** H
7: **end if**
8: Let T and S be the lists of sines and tangents
9: **for** $k = 0$ **to** $N - 1$ **do**
10: **for** $j = 0$ **to** $N - 1$ **do**
11: $m = jk$
12: $H_k = H_k + ROTATION(h_j, N, m, T, S)$
13: **end for**
14: **end for**
15: **return** H

We implemented the DFT algorithm using two for loops for a total of N^2 loops. For each loop we perform $\Theta(1)$ operations, for a total of $\Theta(N^2)$).

6 FFT: recursive implementation and analysis

For large N the DFT algorithm is extremely inefficient. The term Fast Fourier Transform (FFT) is used to describe a computer algorithm that reduces the amount of computational time enormously by taking advantage of the special properties of the complex roots of unity. The basic idea of the FFT is to split the sum

$$H_k = \sum_{j=0}^{N-1} h_j W^{kj}$$

into two sums, each of which is an N/2 point DFT:

$$H_k = \sum_{j=0}^{\frac{N}{2}-1} h_{2j}(W^2)^{jk} + W^k \sum_{j=0}^{\frac{N}{2}-1} h_{2j+1}(W^2)^{kj} := H_k^0 + W^k H_k^1$$

where H_k^0 is the k-th element of the N/2 point DFT of $(h_{2j})_{j=o}^{N/2-1}$, the elements with even index, and H_k^1 is the k-th element of the N/2 point DFT of $(h_{2j+1})_{j=o}^{N/2-1}$, the elements with odd index. This holds for $k < N/2$; if $k \geq N/2$ the two sums are the same as for $k' = k - N/2$ but the weight becomes $W^k = W^{k'}W^{N/2} = -W^{k'}$. Therefore we can write the recursive formula:

$$H_k = H_k^0 + W^k H_k^1, \qquad H_{k+N/2} = H_k^0 - W^k H_k^1$$

for $k = 0, 1, ..., N/2 - 1$. This formula leads immediately to the following recursive implementation of the FFT:

Algorithm 3 FFT_RECURSIVE(h)

Require: h list of complex numbers

```
N ← h.length
if N=2 then
    return h {BASE CASE}
end if
Let H be a list on length N
Let T and S be the lists of sines and tangents
Let h_even and h_odd be the two vectors with even/odd indexes of h
H0 ← FFT_RECURSIVE(h_even)
H1 ← FFT_RECURSIVE(h_odd)
for k = 0 to N/2 − 1 do
    rot ← ROTATION(H1_k, N, k, T, S)
    H_k = H0_k + rot
    H_{k+N/2} = H0_k − rot
end for
return H
```

Note that the base case is with N=2 and not $N = 2^0 = 1$ because the procedure ROTATION cannot be applied for $N < 4$. The analysis of the algorithm leads to the following recurrence for the running time T(N):

$$T(N) = 2T(N/2) + \Theta(N)$$

where $\Theta(N)$ covers the costs of the N/2 rotations, the computations of sines and tangents and the last N additions. By the master theorem $T(N) = \Theta(Nlog_2N)$.

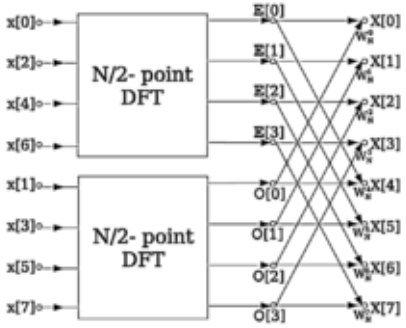
However, as usually happens with recursive algorithms, the space complexity of this implementation is not very good. The function is called twice and for each of the log(N) recursions two reduced FFTs are, which take up actual memory, are added to the call stack. The iterative implementation will overcome this problem.

7 Decimation in Time, radix 2, FFT

The iterative implementation of the FFT starts from the equation

$$H_k = H_k^0 + W^k H_k^1, \qquad H_{k+N/2} = H_k^0 - W^k H_k^1$$

for $k = 0, 1, ..., N/2 - 1$. These two computations together are called a "butterfly"; the set of butterfly computations which compute the DFT of length N from two DFTs of length N/2 can be visualized as in the following diagram for N=8:



The splitting of (H_k) into the two DFTs (H_k^0) and (H_k^1) of size N/2 can be repeated on (H_k^0) and (H_k^1) themselves:

$$H_k^0 = H_k^{00} + (W^2)^k H_k^{01}, \qquad H_k^0 = H_k^{00} - (W^2)^k H_k^{01}$$
$$H_k^1 = H_k^{10} + (W^2)^k H_k^{11}, \qquad H_k^1 = H_k^{10} - (W^2)^k H_k^{11}$$

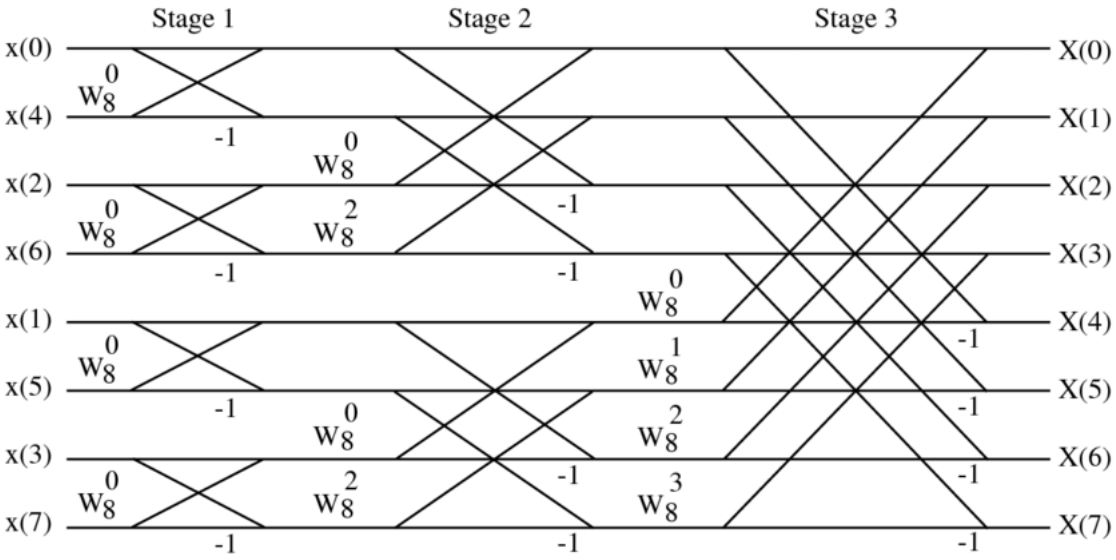
for $k = 0, 1, ..., N/4-1$. Note that (H_k^{00}) is the N/4 point DFT of $\{h_0, h_4, ..., h_{N-4}\}$ (the elements whose indices are equal to 0 module 4); (H_k^{01}) is the N/4 point DFT of $\{h_2, h_6, ..., h_{N-2}\}$ (the elements whose indices are equal to 2 module 4); (H_k^{10}) is the N/4 point DFT of $\{h_1, h_5, ..., h_{N-3}\}$ (the elements whose indices are equal to 1 module 4); (H_k^{11}) is the N/4 point DFT of $\{h_3, h_7, ..., h_{N-1}\}$ (the elements whose indices are equal to 3 module 4). This concept is easier to represent if we consider the binary expansions of the indices $\{0, 1, 2, .., N-1\}$; two elements which are equal module 4 have the same two last digits, therefore:

$$\begin{aligned} (H_k^{00}) &= DFT(h_{...00}) \\ (H_k^{01}) &= DFT(h_{...10}) \\ (H_k^{10}) &= DFT(h_{...01}) \\ (H_k^{11}) &= DFT(h_{...11}); \end{aligned} \tag{1}$$

we have 4 "blocks" of smaller size N/4 DFTs ordered as in 1. Notice that the order follows a reversal of the last two digits (bits) in the binary expansions of

the indices. If we continue this process and we order the "blocks" of smaller DFTs as to compute the butterflies, after $R = log_2 N$ steps we will reach a point where have N blocks of one element. A one point DFT is the identity. Moreover, since the process of indexing the smaller size DFTs by superscripts written by reversing the order the the last bits in the indices will continue, by the time the one point stage (when we begin our computation) is reached all bits in the binary expansions of j will be arranged in reverse order: to begin the first set of butterflies we must first rearrange $\{h_j\}$ so it is listed in bit reverse order.

Here is an example with N=8 using a butterfly diagram ($W_8 = e^{-i2\pi/8}$):



8 Bitreversal Algorithms

In this section we will analyze three different "bitreversal" algorithms: they take as input a list of numbers h of length $N = 2^R$ and return a bit reversal reordering of the initial data with respect to their indices. The first is a brute force algorithm and represents the most intuitive way to perform such a permutation:

- Be $N = 2^R$ the length of a sequence h.
- Form a vector with all the integers from 0 to N-1.
- Convert them in binary expansion and invert each list of bits. This takes $\Theta(log_2(k))$ time for each k=0,...,N-1

- Reconvert the new sequence in decimal position representation. This takes $\Theta(Nlog_2(N))$ operations since we represent each integer using $log_2(N)$ bits.

- Create a copy of h and insert in the new array the elements of h according to the bit reversed indexes. This will take N operations of assigning the computed values to a new array of size N.

In total the computational time of this implementation is $\Theta(Nlog_2(N))$.

The second algorithm is Buneman's; we will consider now only the list of the indexes and not the actual sequence we want to permute. The algorithm is based on a simple property of binary expansions: if we have permuted the N numbers $\{0, 1, ..., N-1\}$ by bit reversing their binary expansions, then the permutation of the 2N numbers $\{0, 1, ..., 2N-1\}$ is obtained by doubling the numbers in the previous permutation to get the first N numbers and then adding 1 to these doubled numbers to get the last N numbers.

Suppose $m = (a_1, a_2, ..., a_R)_{base2}$ where each a_j is either 0 or 1. The number m is mapped to its bit reversed image $P_N(m) = (a_R, ..., a_1)_{base2}$. If we double $P_N(m)$ then

$$2P_N(m) + 1 = (a_R, ..., a_1, 1)_{base2}$$

which is the bit reversed image of

$$m = (0, a_1, ..., a_R)_{base2} \tag{2}$$

where m is considered an element of $\{0, 1, ..., 2N-1\}$. Note that 2 describes the first N numbers in $\{0, 1, ..., 2N-1\}$. Furthermore:

$$2P_N(m) = (a_R, ..., a_1, 0)_{base2}$$

is the bit reversal of

$$m = (1, a_1, ..., a_R)_{base2} \tag{3}$$

which accounts for the last N numbers.

We have implemented Buneman's algorithm through the procedure BITREVERSE2, which takes in input a radix 2 number N and returns an array with the bitreversal permutation of 0,1,.., N-1.

The analysis of the algorithm leads to the following recurrence for the running time T(N):

$$T(2) = \Theta(1), \quad T(N) = T(N/2) + N \ \ if N > 2$$

Algorithm 4 BITREVERSE2(N)

Require: $N = 2^R$

1: **if** N=2 **then**
2: **return** 0,1 {BASE CASE}
3: **end if**
4: $half1 \leftarrow 2 * BITREVERSE2(N/2)$
5: $half2 \leftarrow half1 + 1$
6: **return** half1,half2

By the master theorem we get $T(N) = \Theta(N)$. If we want to apply this approach to a sequence h of any numbers we have first to create a copy of h and then assign to each element of h the correct position according to the results of BITREVERSAL2. This will cost extra N operations for assigning the correct position to the elements.

Buneman's method has the defect that it performs unnecessary swaps: we will now describe a more efficient algorithm that performs only those swaps that are absolutely necessary.

Let's suppose that $N = 2^R$ where R=2Q even. A number m from the list $\{0, 1, ..., N-1\}$ has a binary expansion:

$$m = (a_Q, ..., a_1, b_Q, ..., b_1)_{base2} \Rightarrow P_N(m) = (b_1, ..., b_Q, a_1, ..., a_Q)_{base2} \quad (4)$$

If we let $M = 2^Q = N^{1/2}$, $K = (a_Q, ..., a_1)_{base2}$ and $L = (b_Q, ..., b_1)_{base2}$, then 4 can be written as

$$m = KM + L \Rightarrow P_N(m) = P_M(L)M + P_M(K)$$

for $K, L = 0, 1, ..., M-1$; we are splitting the bit reversal permutation P_N into two bit reversal permutations of square root size P_M. Using $P_M(P_M) = Id$ and P_M is a one to one function on $\{0, 1, ..., M-1\}$ we can express the set of numbers $\{0, 1, ..., N-1\}$ and their bit reversals in the following way:

$$n = P_M(K)M + L \Rightarrow P_N(n) = P_M(L)M + K \quad (5)$$

for $K, L = 0, 1, ..., M-1$. However each swap of data $h_n \Leftrightarrow h_{P_N(n)}$ covers two of the bit reversal described in 5, $n \rightarrow P_N(n)$ and $P_N(n) \rightarrow n$. Moreover we don't need to swap the elements for which $P_N(n) = n$. There are M elements of such type, all the numbers $(a_1, ..., a_Q, a_Q, ..., a_1)_{base2}$: in the representation of eq. 5 are those numbers for which $K = L$. Subtracting these $N^{1/2}$ from the N total elements and multiplying by 1/2 (since there is just one swap needed for each pair swapped) we get $1/2(N - N^{1/2})$ total swaps. It is

possible to prove (Algebra Fundamental Theorem) that these distinct swap are performed by eq 5 when $L < K$ i.e. all the bit reversal are described for $K = 1, 2, .., M - 1$ and $L = 0, 1, ..., K - 1$.

If $R = 2Q + 1$ is odd, a general bit reversal is:

$$m = (a_Q, ..., a_1, c, b_Q, ..., b_1)_{base2} \Rightarrow P_N(m) = (b_1, ..., b_Q, c, a_1, ..., a_Q)_{base2}. \tag{6}$$

Since R-1 is even, using the previous case for $M = (N/2)^{1/2}$, we can express 6 as :

$$\begin{aligned} n = P_M(K)(2M) + L \Rightarrow P_N(n) &= P_M(L)(2M) + K \\ n + M \Rightarrow P_N(n + M) &= P_N(n) + M \end{aligned}$$

for $K = 1, 2, .., M - 1$ and $L = 0, 1, ..., K - 1$. The first row describes all the permutations of the numbers with c=0, while the second row describes all the permutations of the numbers with c=1.

We implemented this in the computer procedure BITREVERSE3, which takes in input an array h of $N = 2^R$ numbers and returns the permuted sequence:

Algorithm 5 BITREVERSE3(h)

1: **if** R=1 **then**
2: **return** h {BASE CASE}
3: **end if**
4: $H \leftarrow$ copy of h
5: **if** R%2=0 **then**
6: $M \leftarrow 2^{R/2}$
7: $PM \leftarrow BITREVERSE3(0 : M-1)$
8: **for** $K = 1$ **to** $M-1$ **do**
9: **for** $L = 0$ **to** $K-1$ **do**
10: $n = PM_K * M + L$
11: $nrev = PM_L * M + K$
12: $H_n \leftarrow h_{nrev}$
13: $H_{nrev} \leftarrow h_n$
14: **end for**
15: **end for**
16: **return** H
17: **end if**
18: **if** R%2=1 **then**
19: $M \leftarrow 2^{(R-1)/2}$
20: $PM \leftarrow BITREVERSE3(0 : M-1)$
21: **for** $K = 1$ **to** $M-1$ **do**
22: **for** $L = 0$ **to** $K-1$ **do**
23: $n = PM_K * (2M) + L$
24: $nrev = PM_L * (2M) + K$
25: $H_n \leftarrow h_{nrev}$
26: $H_{nrev} \leftarrow h_n$
27: $H_{n+M} \leftarrow h_{nrev+M}$
28: $H_{nrev+M} \leftarrow h_{n+M}$
29: **end for**
30: **end for**
31: **return** H
32: **end if**

If R is even the computational time is $T(N) = T(N^{1/2}) + \sum_{K=1}^{M-1} \sum_{L=0}^{K-1} \Theta(1) = T(\sqrt{N}) + \Theta(N))$ and $T(N) = \Theta(1)$ if N=1.

If R is odd similarly $T(N) = T(\sqrt{N/2}) + \Theta(N))$. Therefore, if we set $R = log_2 N$ we can write the asymptotical computational time with respect

to R:

$$T(R) = T(\left\lfloor \frac{R}{2} \right\rfloor) + \Theta(2^R) = c2^R + c2^{\lfloor \frac{R}{2} \rfloor}) + ... + 2 + 1 = c\sum_{k=0}^{logR} 2^{2^k} + 1$$

where c is a constant. We guess that the solution is $O(2^R) = O(N)$ which can be easily verified through the substitution method . The computational time of BITREVERSE3 is equivalent to Buneman's algorithm in terms of asymptotic behavior; nevertheless it is immediate to see that there is a considerable time saving because reduces to one half the number of permutations and also the number of recursive calls is $log_2(log_2(N))$ against $log_2(N)$ calls of BITREVERSE2.

9 The Butterflies' Implementation

In this section we will see in detail how we implemented the computation of butterflies: we compute N/2 of them for R iterations for a total of Nlog(N). Suppose we are at stage i of the computation, starting from 0 (first stage) to R-1 (last stage). Each of the 2^{R-i-1} new blocks will be composed by a total of 2^{i+1} elements (the double of the 2^i elements in each of the old blocks) obtained through 2^i butterfly computations using the elements of the previous two consecutive blocks in the same position of the new one (they are ordered with respect two the superscript indexes we explained in Section 7): the first one is with the elements whose superscript index finishes with 0 and the second one with 1. For example at stage 1 we will have N/4 new blocks of length 4, each of them computed using two old consecutive blocks of length 2. At the last stage R-1 we will have to computed one last block of length N using the two old blocks of length N/2: this last block is our DFT sequence. These equations describe the computation of a new block starting from two consecutive old blocks by considering them as entities apart, for fixed stage $i \in \{0, 1, ..., R-1\}$:

$$H_k^{new} = H_k^{..0} + (W^{2^{R-i-1}})^k H_k^{..1}, \qquad H_{k+2^i}^{new} = H_k^{..0} - (W^{2^{R-i-1}})^k H_k^{..1}$$

for $k = 0, 1, .., 2^i - 1$. Actually the old blocks have a precise position and we have to detect them using the indexes start, middle, end. Since we know their number and their length this is not a difficult task.

We implemented this method in the procedure FFT:

Algorithm 6 FFT(h)

Require: h list of complex numbers

$N \leftarrow h.length$
$R \leftarrow log_2(N)$
$h \leftarrow BITREVERSE3(h)$
Let T and S be the lists of sines and tangents
for stage=0 **to** R-1 **do**
  $futureblocks \leftarrow 2^{R-stage-1}$ {number of future blocks}
  $length \leftarrow 2^{stage}$ {length of half future block, i.e. of one current(old) block}
  for L=0 **to** futureblocks **do**
    $start \leftarrow L * 2length$ {L identifies which block the algorithm is currently working on}
    $middle \leftarrow start + length - 1$
    $end \leftarrow middle + length$
    $top \leftarrow$ new empty array
    $bottom \leftarrow$ new empty array {now start a loop which spans the first block}
    for k=start **to** middle **do**
      $m \leftarrow 2^{futureblocks} * k$ {exponent of W}
      $rot \leftarrow ROTATION(h[k + length], N, m, T, S)$ {the elements of the second block are rotated}
      top.append(h[k]+rot) {upper part of the new block}
      bottom.append(h[k]-rot) {lower part}
    end for
  end for
  $h[start, ..., middle] \leftarrow top$
  $h[middle + 1, ..., end] \leftarrow bottom$
end for
return h

The computational time is $\Theta(Nlog(N))$.

10 Computing a real FFT

In this section we shall take advantage of some properties of the DFT which arise when the data are real to reduce the number of calculations involved of about one half.

First we present an algorithm to compute two real FFT simultaneously. Suppose $\{f_j\}_{j=0}^{N-1}$ and $\{g_j\}_{j=0}^{N-1}$ are sequences of N real numbers.

Define $h_j = f_j + ig_j$: taking the N point DFT we get $H_k = \sum_{j=0}^{N-1}(f_j + ig_j)W^{jk} =: F_k + iG_k$ where F and G are FFTs of the real sequences. There is a crucial symmetry property we need to use: is we take the complex conjugate F^*_{N-k} of F_{N-k} we get

$$F^*_{N-k} = \Big[\sum_{j=0}^{N-1} f_j W^{j(N-k)}\Big]^* = \sum_{j=0}^{N-1} f_j^*(W^{jN})^*(W^{-jk})^* = F_k$$

because f_j is real, W is N periodic and $(W^{-jk})^* = W^{jk}$. Therefore we get $H^*_{N-k} = F^*_{N-k} - iG^*_{N-k}F_k - iG_k$ and so, combining with $H_k = F_k + iG_k$, we get

$$F_k = \frac{1}{2}[H^*_{N-k} + H_k], \quad G_k = \frac{i}{2}[H^*_{N-k} - H_k]. \tag{7}$$

We have shown how to to obtain two real FFTs with the same (asymptotical) cost of performing one complex FFT using a three step method:

1. Form the sequence $h_j = f_j + ig_j$
2. Perform the N-point FFT of h
3. Use 7 to obtain F_k and G_k

Now we use this method to compute the FFT of a real sequence; consider the first reduction involved in computing the decimation in time $F_k = F^0_k + F^1_k W^k$, F^0 and F^1 are the N/2-pint FFTs of the even and odd index subsequences. To compute them together we form the sequence

$$h_j = f_{2j} + if_{2j+1} \quad (j = 0, ..., N/2)$$

and we get

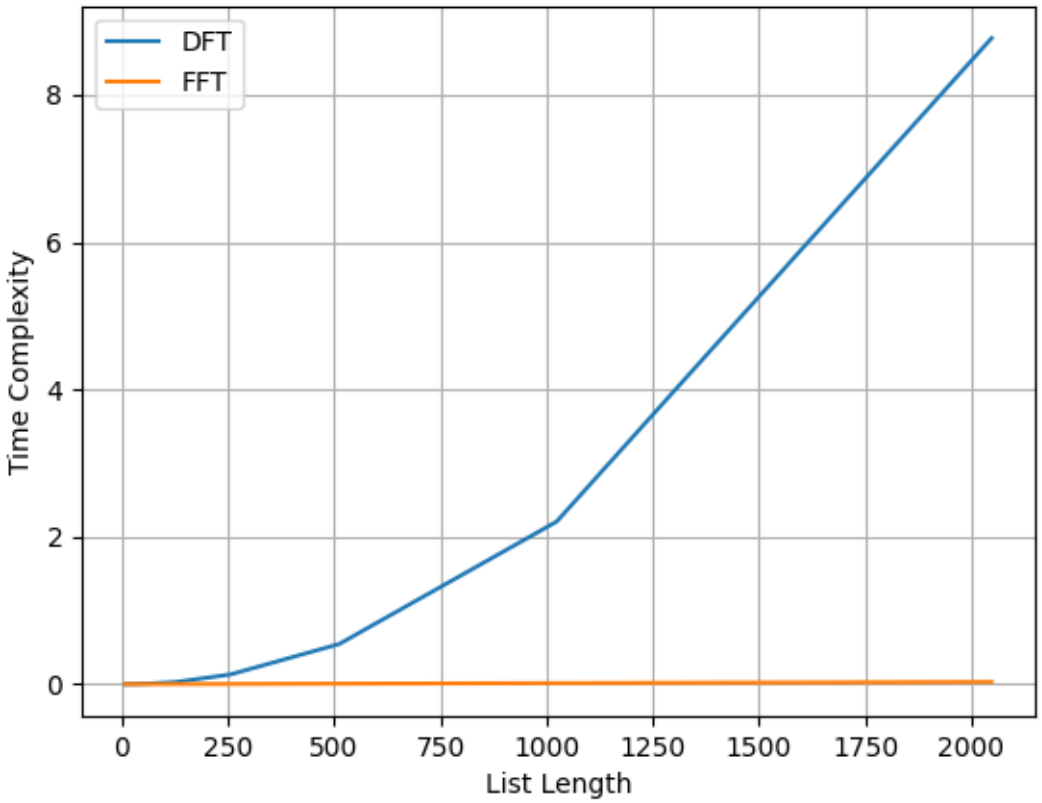
$$F^0_k = \frac{1}{2}[H^*_{N/2-k} + H_k], \quad F^1_k = \frac{i}{2}[H^*_{N/2-k} - H_k]$$

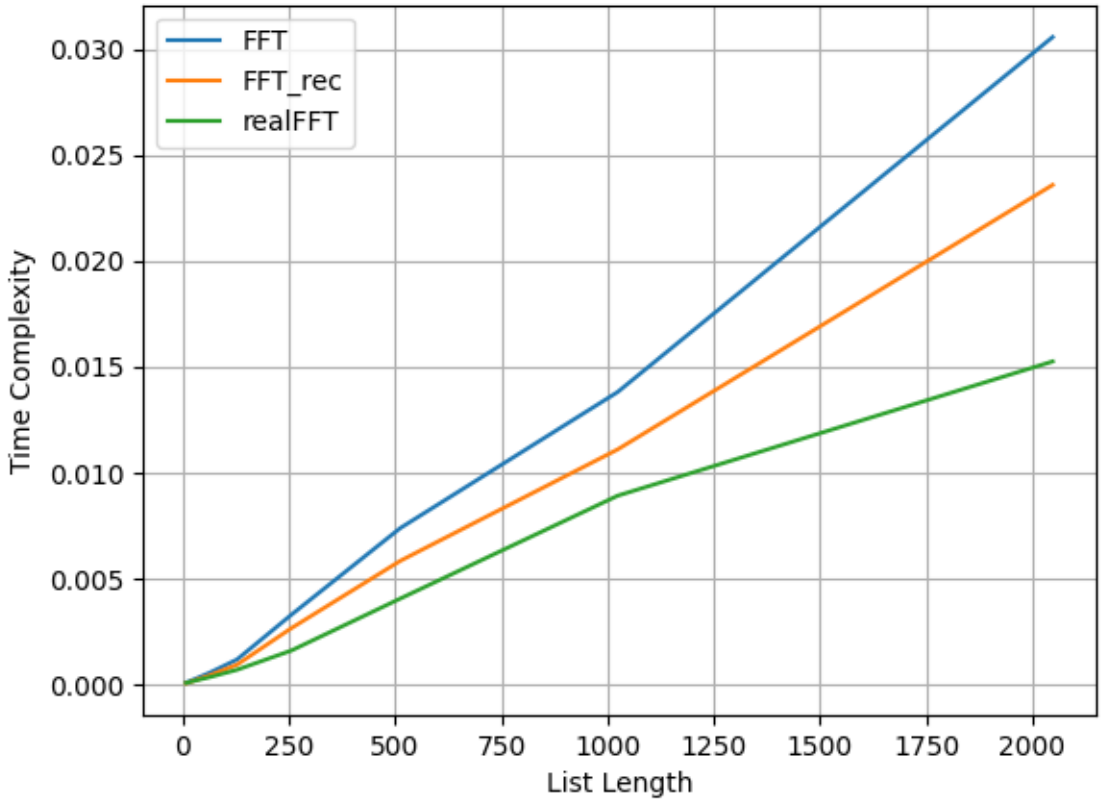
for $k = 0, .., N/2 - 1$. We then perform the last set butterflies to compute the complete DFT.

The computational cost of this operation is $\Theta(Nlog(N/2)$ and it can be seen that the coefficient of the asymptotic limit is about one half the one for a simple FFT.

11 Testing on real data

We tested the four algorithms DFT, FFT, FFT_RECURSIVE and REALFFT described above on sequences of random integers ranging from 0 to 2^{14} for $N = 2^R$ with R=3,...,11. The two plots compare the running times: the first one shows how faster it is an FFT algorithm compared to a DFT one; the second plot proves that the REALFFT procedure actually saves time compared to the other two asymptotically equivalent FFT procedures.





12 Computing Polynomial Multiplication using FFT

Digital Signal Processing (DSP) is one of the major applications of DFT: we apply the transform to a the sequence $(h_k)_{k=0}^{N-1}$ which is meant to be a sampling of a signal at N time intervals. Polynomials are important in DSP because calculating the DFT can be viewed as a polynomial evaluation problem and convolution can be viewed as polynomial multiplication.

Indeed a length N sequence can be represented by an N-1 degree polynomial A defined by:

$$A(x) = \sum_{k=0}^{N-1} h_k x^k$$

This polynomial is a single entity with coefficients being the values of h_k (in implementations this is the same as representing polynomial as arrays, with

the position of an element indicating the coefficient degree). It is immediate to see that the N-points DFT of the sequence is nevertheless the evaluation of A in the N different complex roots of the unit. It is well known that a polynomial (with complex coefficients) is uniquely identified by its values in N points. Therefore it is intuitive that the DFT is invertible:

Definition 12.1. *If $(H_k)_{k=0}^{N-1}$ is a sequence of N complex numbers, then its N-points inverse DFT is defined by*

$$h_k = \frac{1}{N} \sum_{j=0}^{N-1} H_k W^{kj}$$

for $k = 0, 1, ..., N-1$, where in this case $W = e^{i2\pi/N}$ is the principal complex root of the unity.

The implementation of the procedure FFT_INVERSE(h) is identical to the FFT except for the fact that the rotations are performed according to a rotation anti-clockwise so the signs change: the inverse is indeed an FFT with weight $W = e^{i2\pi/N}$ instead of $e^{-i2\pi/N}$.

Consider now the sequence $(h_k)_{k\in\mathbb{Z}}$ where $h_k = h_{k_{modN}}$: it is the periodic extension the N-point sequence h.

Definition 12.2. *Let $(a_k)_{k\in\mathbb{Z}}$ and $(b_k)_{k\in\mathbb{Z}}$ be two sequences having period N. Their cyclic convolution is defined by:*

$$(a * b)_k = \sum_{j=0}^{N-1} a_j b_{k-j}$$

for $k \in \mathbb{Z}$. Note that it has again period N.

Theorem 12.1. *Convolution Theorem The N point DFT of $((a * b)_k)_{k=0}^{N-1}$ is the component-wise product of the two DFTs A and B of a and b: $(A_k B_k)_{k=0}^{N-1}$*

If $A(x) = \sum_{k=0}^{N-1} a_k x^k$ and $B(x) = \sum_{k=0}^{N-1} b_k x^k$ are two polynomials their product is the polynomial $C(x) = \sum_{k=0}^{2N-1} c_k x^k$ where $c_k = \sum_{j=max\{0,k-N+1\}}^{min\{k,N-1\}} a_k b^{k-j}$ (Note that we didn't specify A and B to have degree N-1 therefore $a_{N-1}$and b_{N-1} can be equal to 0) The 2N-1 coefficients of C are also called the Linear Convolution of the two sequences a and b which can be expressed as a 2N cyclic convolution of the sequences a' and b' obtained from a and b adding N zeros and expanding them over $\mathbb{Z}$ using the period 2N: $c_k = (a' * b')_k$.

The computation of the polynomial product (equivalent to a linear convolution)takes time $\Theta(N^2)$ since we have to multiply each coefficient of A by

Algorithm 7 FOURIERMULTIPLY(p1,p2)

1: $n1, n2 \leftarrow len(p1), len(p2)$
2: $R \leftarrow log_2(max(n1, n2) + 1$
3: $N = 2^R$
4: append 0 to p1 and p2 until I reach length 2N
5: $f1 \leftarrow FFT(p1)$
6: $f2 \leftarrow FFT(p2)$
7: $f3 \leftarrow f1 * f2\ component - wise$
8: **return** FFT_INVERSE(f3)

each coefficient of B. The convolution theorem and the FFT algorithm give us a way to perform it in $\Theta(Nlog(N)^)2$:

This is a time-efficient way of performing polynomial multiplication: the standard implementation would take time $O(N^2)$; using the FOURIERMULTIPLY procedure the computational time needed is $3 * \Theta(2Nlog_2(2N) = \Theta(Nlog_2(N)^2$ since we call three times a FFT of length 2N. Moreover if the coefficients are real we can perform the first two FFTs simultaneously. It is not always the same with respect to memory efficiency; indeed the Implementation of a polynomial using an array needs a lot of memory if the polynomial of degree N has many lower degree coefficients equal to 0: for example we need an array of length 5 to store both x^4 and $1 + x + x^2 + x^3 + x^4$. Moreover we have to perform the two FFTs and the inverse over arrays of length 2N: if N is big we need to store a lot of redundant data.

In our case, since we implemented only the radix 2 FFT, we can store polynomials only in arrays of radix 2 length which is a great waste of memory. We will now implement a polynomial class together with a method to compute the multiplication using linked lists.

13 Linked List Polynomial Multiplication

We implemented a class Poly using singly linked lists: each node represents a monomial and has 3 attributes representing the coefficient, exponent and a pointer to next node. An object from Poly is a lists of nodes with coefficients different from zero and sorted from the higher to the lower degree term. The head is therefore the node with the highest exponent and determines the degree of the polynomial. We implemented two functions to sum and multiply two Poly objects: the peculiarity of this implementation is that it is very efficient (both in space and memory) for polynomials with high degree and a very low number of terms; the drawback is that we don't have random access

to the elements of the polynomial. The running times of the functions sum and multiply vary a lot depending on the length of the polynomial: in the best case can be $\Theta(1)$ while in the worst case multiply can be (N^3).

See file Jupyter Notebook for the implementation.

We analyzed the two algorithms for polynomial multiplication, the FOURIERMULTIPLY and the one implemented for objects in the class Poly, for two different kind of polynomials in order to show the advantages and disadvantages of the two implementations. First we measured the running time of the two algorithms multiplying two binomials in the form $ax^{145} + b$ where a and b are random integers. In this case the linked list multiplication performed better than the fourier's because this type of polynomials is implemented more efficiently using linked lists.

Then we measured the running time multiplying two polynomials with degree 145 but with random integer coefficients. This time the fourier approach is much faster.

BINOMIAL PERFORMANCE: fourier time 0.09422421455383301 linked list time 6.389617919921875e-05

RANDOM POLYNOMIALS PERFORMANCE: fourier time 0.01749706268310547 linked list time 5.300473928451538

Riferimenti bibliografici

[1] James S. Walker. *Fast Fourier Transforms*. 1996.

[2] C. Sidney Burrus, Matteo Frigo, Steven G. Johnson, Markus Pueschel, Ivan Selesnick *Fast Fourier Transforms*. 2012.