Fast Fourier Transform

Saurabh Sood

Abstract

This paper presents the implementation and analysis of the Fast Fourier Transform. The Fast Fourier transform algorithm is a major asymptotic improvement over the Discrete Fourier transform, and has varied applications in the field of signal processing. The Discrete Fourier Transform (DFT) has a time complexity of $O(n^2)$, whereas the Fast Fourier Transform runs in $O(nlog_2n)$. The paper looks at an implementation of the FFT, and provides a rigorous mathematical analysis of the time, and space complexities in the Worst, Average and Best case inputs.

1 Introduction

The Discrete Fourier Transform (DFT) is the most important transformation used in Fourier Analysis. It is used to convert a set of function values into a combination of periodic functions (namely sine waves). More intuitively, it can be thought of as converting a function from the time domain to the frequency domain. This has various applications in the field of Image Processing, Digital Signal Processing to name a few. In the field of Digital Signal Processing, it can be used to find the various frequencies in a noisy channel, whereas in Image Processing, it can be used to extract the actual image from a blurred image, by clearing out the noisy pixels. In Mathematics, it can be used to solve Partial Differential Equations really fast. It is such an important algorithm, that special hardware is built so that it runs really fast. In hardware, the DFT is usually implemented with the Fast Fourier Transform (FFT) algorithm, which is an asymptotically faster algorithm for the DFT.

2 History

The DFT can be traced to the work done by Fourier, who proposed that a function could be represented as a combination of sinusoidal functions. However, the algorithm to compute the DFT was inefficient, and could not run on most hardware at that time. There was work done by Gauss for a fast running implementation of the DFT, but it was never published [2]. Currently, the most commonly used version of the FFT can be traced to the work done by Cooley, an IBM engineer, and Tukey, a statistician. Their work involved improving the algorithm, so that it works when their input size is not a power of 2 as well.

3 The Discrete Fourier Transform

3.1 Complex Roots of Unity

A complex n^th root of unity is a complex number ω such that

$$\omega^n = 1$$

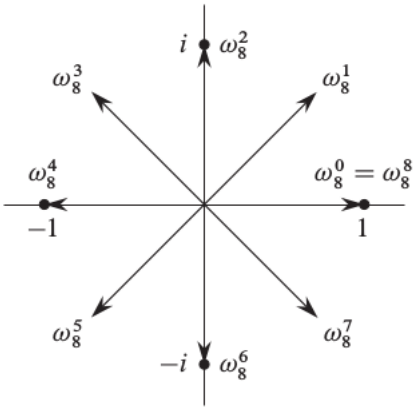
The n complex roots of unity can be represented as follows:

$$e^{\frac{2\pi ik}{n}}; \text{ for } k = 0, 1, 2...n-1$$

This can be represented trigonometrically as:

$$e^{iu} = cos(u) + isin(u)$$

Pictorially, the roots of unity can be represented as shown in the following figure:



where the values of $w_8^0...w_8^7$ are represented in the complex plane, and $w_8 = e^{\frac{2\pi_1}{8}}$ is the principal 8^{th} root of unity

3.2 Cancellation Lemma

$$\omega_{dn}^{dk} = \omega_n^k; \text{ for } n \geq 0, k \geq 0, d > 0$$

This follows directly from the definition of the complex roots of unity:

$$\omega_n = e^{\frac{2\pi i}{n}}$$

3.3 Halving Lemma

$$(\omega_n^{\frac{k+n}{2}})^2 = (\omega_n^k)^2; \text{ if } n > 0 \text{ is even}$$

This implies that for an even value of n, the squares of the n complex roots of unity are the $\frac{n}{2}$ complex $(\frac{n}{2})^{th}$ roots of unity

3.4 Summation Lemma

$\sum_{j=0}^{n-1}(\omega_n^k)^j = 0$; for $n \geq 1$, and for k not divisible by n

The increasing powers of complex roots of unity sum to 0

3.5 The Algorithm

The Discrete Fourier transform can be represented with the following conversion:

$$\begin{aligned} y_k &= A(w_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj} \end{aligned}$$

for $k = 0, 1, 2..., n-1$, and a_j is the coefficient matrix of a polynomial $A(x)$

$$A(x) = \textstyle\sum_{j=0}^{n-1} a_j x^j$$

The vector $y = (y_0, y_1 ... y_{n-1}$ is called the *Discrete Fourier Transform* of the coefficient vector $a = (a_0, a_1, ..., a_{n-1})$

3.6 Pseudocode

Algorithm 1: $O(n^2)$ implementation of the Discrete Fourier Transform

```
Data: An input coefficient vector a
Result: A vector y, of length n, which is the Discrete Fourier Transform
        of the given input coefficient vector a
initialization;
n = a.length ;where n is a power of 2
if n==1 then
    return a
else
    for k = 0 to n do
        sum = 0
        for j = 0 to n − 1 do
            sum+ = a_j e^(−2πijk/n)
        end
        y_k = sum
    end
    return y
end
```

3.7 Analysis of the DFT

Time Complexity
The above algorithm for computing the Discrete Fourier Transform has a time complexity of $O(n^2)$.

- The inner loop computes the k_{th} coefficient of the output vector. Mathematically, it represents:

  $$y_k = \sum_{j=0}^{n-1} a_k e^{\frac{-2\pi ijk}{n}}$$

  This takes n iterations

- The outer loop computes the coefficient for each of the n input vectors. This also takes n iterations.

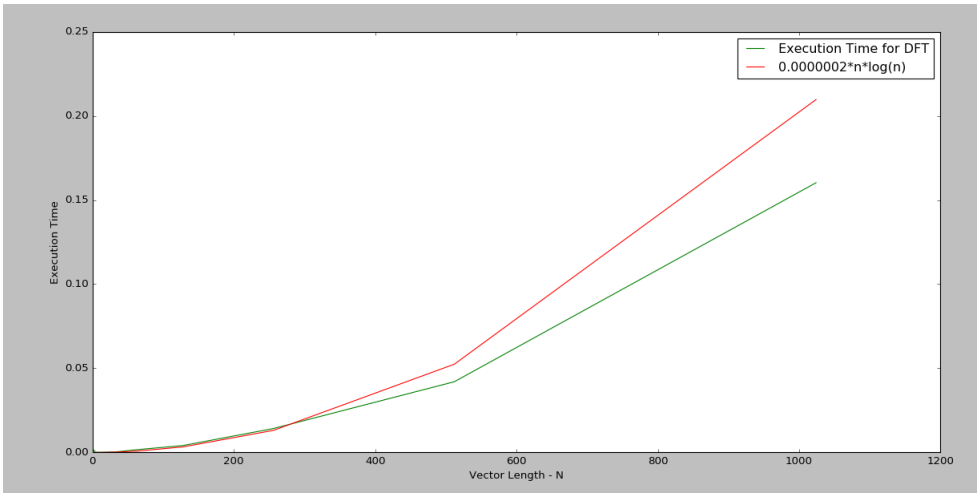
Thus, the overall time complexity is $O(n^2)$

Runtime Analysis for Best Case, Worst Case, Average Case Inputs

- **Worst/Average Case**
  There is no worst case input for the DFT algorithm. In the average case, the input vector is of size n. The DFT in this case would return a vector of size n. The runtime complexity of the algorithm in this case would be $O(n^2)$.

- **Best Case**
  The Best Case of the DFT is the same as the average case. The runtime in this case will also be $O(n^2)$.

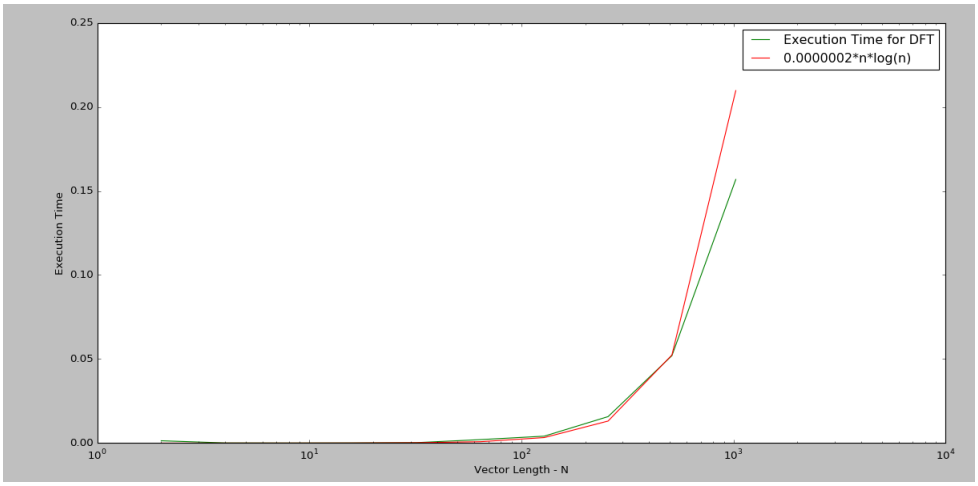
Numerical Characterization of the Running Time



Plot showing the runtime of DFT.

The input here is a vector of length 2-1024 (Due to limitations in RAM, it could not be extended for further values of N). The y-axis of the graph represents the running time of the algorithm. Here, the input vector a is randomly generated. The graph shows a $O(n^2)$ bound for the runtime of the algorithm.

The graph for the running time for the DFT in semilogx scale is as shown below:



Space Complexity

- The input and output vectors are of length n each. Both take $O(n)$ space.
- The intermediate *sum* variable takes $O(1)$ space

Thus, the overall space complexity of the algorithm is $O(n)$

4 Fast Fourier Transform

[3] The Discrete Fourier Transform(FFT) runs in $O(n^2)$ running time, where n is the polynomial vector length. The algorithm showed earlier doesn't scale well for very high powers of n. The Fast Fourier Transform algorithm takes advantage of special properties like symmetry, in the computing the roots, and reduces the computational time taken for computing the Discrete Fourier Transform. The runtime complexity of the FFT is $O(nlogn)$, as opposed to $O(n^2)$ in the base version of the DFT.

4.1 The Algorithm

For the purpose of the paper, we assume that n is a power of 2. The FFT works for non-powers of 2, but the assumption makes the analysis easier. The FFT

is a divide and conquer algorithm. It separately considers the even and odd coefficients of $A(x)$

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + ... + a_{n-2}x^{\frac{n}{2}-1}$$
$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + ... + a_{n-1}x^{\frac{n}{2}-1}$$

where $A^{[0]}(x)$ considers the even coefficients of $A(x)$, and $A^{[1]}(x)$ considers the odd coefficients of $A(x)$
The problem of computing the DFT can be summarized with the following:

- **Divide Step**
  evaluating the $\frac{n}{2}$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ at the points $(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2$
- **Conquer Step**
  Combine the results using the formula:

$$A(x) = A^{[0]}(x^2) + A^{[1]}(x^2)$$

Using the Halving Lemma, the polynomial evaluations at $(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2$ consist of the $\frac{n}{2}$ complex $\frac{n}{2}^{th}$ roots of unity with each root occuring twice. Thus, we have two subproblems of the same form as the original problem, and are half the size of the original problem. This leads to a recursive algorithm for the FFT, where the original problem of $DFT_{n/2}$ reduces to two problems of $DFT_{n/2}$

4.2 Analysis

Algorithm 2: Recursive Implementation of the Fast Fourier Transform

Data: An input coefficient vector a
Result: A vector of length n, which is the Discrete Fourier Transform of the given input coefficient vector a

```
1  initialization;
2  n = a.length ;where n is a power of 2
3  if n==1 then
4  |  return a
5  else
6  |  ω_n = e^(2πi/n)
7  |  ω = 1
8  |  a^[0] = (a^0, a^2, ..., a^(n-2)
9  |  a^[1] = (a^1, a^3, ..., a^(n-1)
10 |  y^[0] = RECURSIVE_FFT(a^[0])
11 |  y^[1] = RECURSIVE_FFT(a^[1])
12 |  for k = 0 to n/2 - 1 do
13 |  |  y_k = y_k^[0] + ω y_k^[1]
14 |  |  y_(k+n/2) = y_k^[0] - ω y_k^[1]
15 |  |  ω = ω ω_n
16 |  end
17 |  return y
18 end
```

[1]

- Lines 3-4 represent the base case of the recursion, where the DFT of a single element is the element itself
- Lines 8-9 define the coefficient vectors for the polynomials $A^{[0]}$, and $A^{[1]}$.
- Lines 6,7,15 ensures that ω is updated whenever the DFT coefficient vectors get updated in Lines 13,14. Thus, we will always have

$$\omega = \omega_n^k$$

- Lines 10, 11 perform the recursive $DFT_{\frac{n}{2}}$ computations. The for loop runs for $k = 0, 1, 2... \frac{n}{2} - 1$ iterations. So, we have:

$$y_k^{[0]} = A^{[0]}(\omega_{\frac{n}{2}}^k)^2$$
$$y_k^{[1]} = A^{[1]}(\omega_{\frac{n}{2}}^k)^2$$

Since $\omega_{\frac{n}{2}k} = \omega_n^{2k}$ (Cancellation Lemma), we have the following results:

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k})^2$$
$$y_k^{[1]} = A^{[1]}(\omega_n^{2k})^2$$

- Lines 13,14 combine the results of the $DFT_{n/2}$ computations. We get the following results:

$$\begin{aligned} y_k &= y_k^0 + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \end{aligned}$$

- For $y_{\frac{n}{2}}, y_{\frac{n}{2}-1}...y_{n-1}$, setting $k = 0, 1, ..., \frac{n}{2} - 1$ yields:

$$\begin{aligned} y_{k+\frac{n}{2}} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+\frac{n}{2}} y_k^{[1]}; \text{ since } \omega_n^{k+\frac{n}{2}} = -\omega_n^k \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+\frac{n}{2}} A^{[1]}(\omega_n^{2k}); \text{ since } \omega_n^{2k+n} = \omega_n^{2k} \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+\frac{n}{2}}) \end{aligned}$$

  Thus, we get the same vector which is the same as the output of the DFT of the input vector a.

4.3 Runtime Analysis

To find the running time of the *RECURSIVE_FFT* procedure, we note that each invocation (aside from the recursive step) takes $\theta(n)$ time. Thus, we can form the following recurrence:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \Theta(n) \\ &= \Theta(nlgn) \text{ ; Using the master theorem} \end{aligned}$$

Thus, the Fast Fourier Transform is an asymptotic upgrade over the basic implementation of the Discrete Fourier Transform

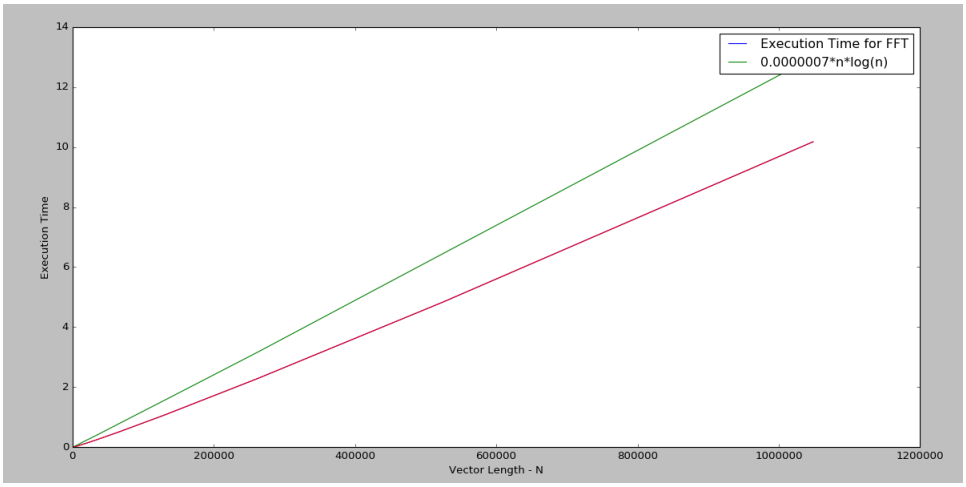
4.4 Runtime Analysis for Worst Case, Average Case Inputs

- **Worst/Average Case**
  There is no worst case input for the FFT algorithm. In the general case,

the coefficient vector is a vector of length n, where $n = 2^k$; $k > 0$.
In this case, the algorithm will take $O(nlogn)$ time.

- **Best Case**
  The Best Case of the DFT is the same as the average case. The runtime in this case will also be $O(n^2)$.

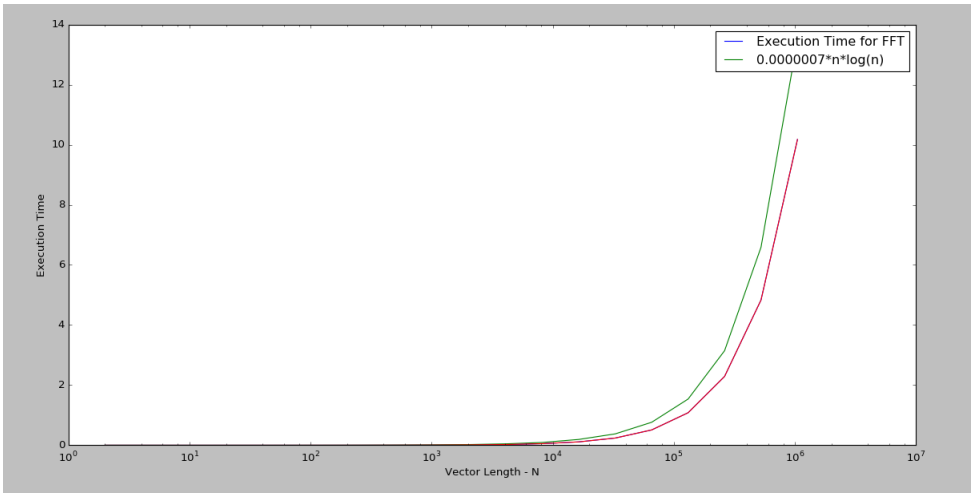
4.5 Numerical Characterization of the Runtime



Plot showing the runtimes of FFT

The input here is a vector of length 2-1048576 (2^0 to 2^{20}). The y-axis of the graph represents the running time of the algorithm. Here, the input vector a is randomly generated. The graph shows a $O(nlog(n))$ bound for the algorithm.

The graph for the running time for the FFT in semilogx scale is as shown below:



4.6 Space Complexity

- The input and output vectors take $O(n)$ space
- The computations and multiplications take $O(1)$ extra space

Thus, the overall space complexity of the algorithm is $O(n)$

4.7 Extensions

The paper explored the Discrete Fourier Transform, it's runtime, and space complexity. It also explored the improvements done with using the FFT algorithm, and showed that it is an asymptotic improvement over the base DFT algorithm.
There are some extensions for the analysis of the DFT and FFT done in the paper, which could be done.

- Due to RAM limitations, the base DFT algorithm could be run for only a small set of values (2^{10}). The algorithm could be run on a faster computer (or a supercomputer) to clearly show the $O(n^2)$ bound for a larger set of values.
- The algorithms presented in the section run for input vectors having length as a power of 2. This makes the algorithm easier to analyze. However, FFT and DFT run on input vectors of arbitrary sizes. For future work, this could be incorporated in the implementation and the analysis.

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[2] John W. Tukey James W. Cooley. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[3] Pythonic Perambulations. Understanding the fft algorithm. `https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/`.