# 1. FFT

## 1.1. Enfoque

Se decidió desarrollar un algoritmo FFT, ya que se contó con conocimiento del campo de ciencias de la computación para resolver problemas de programación dinámica. Se mostrará la derivación de un algoritmo menos eficiente pero más entendible, y luego se describirán sus posteriores optimizaciones. En el resultado final se llegó a un algoritmo con $\frac{n}{2}log_2 n$ multiplicaciones, $6n$ memoria, en otras palabras, $\Theta(nlog(n))$ tiempo, $\Theta(n)$ memoria.

## 1.2. El problema

El cálculo de la DFT consiste en producir una la salida $X[k]$ de coeficientes mediante la entrada $x[n]$, ambos arreglos de igual longitud $N$. La fórmula que describe la DFT que utilizamos es:

$$DFT[X[n]] = X[k] = \sum_{n=0}^{N-1} x(n)e^{-i2\pi kn/N} \tag{1}$$

Se necesita resolver la DFT en tiempo subcuadrático, es decir en menos de $\Theta(n^2)$, que es la solución trivial.

## 1.3. Primera observación de interés

Si la longitud de la sucesión $N$ es par se puede escribir $X[k]$ como:

$$DFT[X[n]] = X[k] = \underbrace{\sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-i2\pi kn/(N/2)}}_{DFT[X[2n]=Y[n]]} + e^{i2\pi k/N} \overbrace{\sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-i2\pi kn/(N/2)}}^{DFT[X[2n+1]=Z[n]]} \tag{2}$$

$$DFT[\underbrace{X[n]}_{0\leq n<N}] = DFT[\underbrace{Y[n]}_{0\leq n<N/2}] + e^{i2\pi k/N} DFT[\overbrace{Z[n]}^{0\leq n<N/2}] \tag{3}$$

Donde $X[2n] = Y[n]$, $X[2n+1], Y[2n+1]$ son sub-sucesiones de $X[n]$ tomando los índices pares e impares respectivamente. Podemos decir que descompusimos el problema de la DFT como dos problemas de DFT con sucesiones de la mitad de tamaño, es decir, conseguimos una solución recursiva. Es importante observar que la única forma de que la solución sea viable es que $N$ sea una potencia de dos, para poder garantizar que al dividir los subproblemas siempre $N_i$ sea par.

## 1.4. Algoritmo conceptual

Si bien la formulación anterior da lugar a la implementación de un algoritmo recursivo "top-down" se procedió a desarrollar una fórmula "bottom-up" para resolver el problema iterativamente, y por lo tanto provocar que la ejecución tenga una constante menor.

Se definió el subproblema $DFT[i][j][k]$ como el $k$-ésimo coeficiente, del subarreglo de longitud $2^i$ que comienza en el casillero $j$. Se puede mostrar mediante la ecuación 3 en esta formulación es equivalente a

$$DFT[i][j][k] = \begin{cases} DFT[i-1][j][k\,\%2^{i-1}] + W[2^i][k]DFT[i-1][j+N/2^i][k\,\%2^{i-1}] & \begin{array}{l} 1 \leq i \leq log_2 N \\ 0 \leq j < N/2^i \\ 0 \leq k < 2^i \end{array} \\ A[j] & \begin{array}{l} i = 0 \\ k = 0 \end{array} \end{cases} \tag{4}$$

Donde $W[N][k] = e^{-i2\pi k/N}$. Con esta precisa formulación se puede implementar con 3 sentencias *for* un algoritmo de $O(nlogn)$ tiempo y $O(n^2logn)$ memoria.

## 1.5. Optimizaciones de memoria

Se presentan dos modificaciones superficiales y técnicas, no conceptuales, que no alteran la lógica escencial del algoritmo, pero que lo vuelven práctico.

### 1.5.1. Optimización 1

Se necesita reducir el orden de memoria de $\Theta(n^2)$, el cual es muy limitante. La primera optimización es mediante la observación de que para calcular $DFT[i]$ solo necesitamos tener previamente calculado $DFT[i-1]$. Por lo tanto podemos calcular en un arreglo el DFT siguiente y en la posterior iteración escribir de dicho arreglo al anterior, "yendo y volviendo". De esta forma se reduce el uso de memoria a $\Theta(n^2)$. Este valor todavia no es conveniente, ya que la expectativa es que tanto el tiempo como la memoria del algoritmo sean subcuadráticos.

### 1.5.2. Optimización 2

La observación necesaria para reducir el uso de memoria es que, en cada iteración $jk < N$, por lo que en realidad es posible almacenar todos los valores en un arreglo de longitud $N$ (es decir, "comprimiendo" la matriz). Como el número de filas y columnas de la matriz es variable para cada valor de $i$ es necesario implementar una lógica que se adapte a este hecho ($mat[i][j] = arr[i * colsize + j]$). De esta forma se redujo la memoria del algoritmo a $\Theta(n)$.

## 1.6. Optimizaciones de tiempo

Se mostrarán dos optimizaciones de tiempo que si bien no mejoran la complejidad del algoritmo en tiempo (de hecho, no son conocidos algoritmos mejores que $\Theta(nlogn)$), reducen, en una constante, su demora.

### 1.6.1. Cálculo $W[n][k] = e^{-i2\pi k/n}$

Calcular cada vez que se necesitan dichos factores si bien no modificaría la complejidad del algoritmo, si impacta en el tiempo total. Para ello se precalcula en el inicio del algoritmo $W[k] = e^{-2i\pi k/N}$ con el $N$ más grande. Como vale que:

$$W[n][k] = e^{-i2\pi K/n} = e^{-i2\pi K/N*N/n} = W[K*N/n] = W[K*N/2^i] = W[K*N/2^i \,\%N] \tag{5}$$

Por lo tanto calculando inicialmente $W[k]$ con $0 \leq k < N$ ya se tiene información para obtener en una sola operación la exponencial correspondiente cuando sea necesario.

### 1.6.2. Optimización Butterfly

Otra optimización importante que permitirá reducir a la mitad la cantidad de multiplicaciones complejas que realiza el algoritmo se denomina optimización "Butterfly". Consiste en observar dos propiedades:

$$W[n][k] = -W[n][k+n/2] \tag{6}$$

$$DFT[..][..][k\,\%2^{i-1}] = DFT[..][..][(k+n/2)\,\%2^{i-1}] \tag{7}$$

La forma de utilizarla consiste en, computar en paralelo las siguientes expresiones:

$$DFT[i][j][k] = DFT[i-1][j][k\,\%2^{i-1}] + W[2^i][k]DFT[i-1][j+N/2^i][k\,\%2^{i-1}] \tag{8}$$

$$DFT[i][j][k+n/2] = DFT[i-1][j][k\,\%2^{i-1}] - W[2^i][k]DFT[i-1][j+N/2^i][k\,\%2^{i-1}] \tag{9}$$

Realizando el producto $W[2^i][k]DFT[i-1][j+N/2^i][k\,\%2^{i-1}]$ una sola vez.

# 2. Conclusión

Se llegó a un algoritmo muy optimizado ($6n$ memoria, $\frac{n}{2}log_2 n$ tiempo) con una complejidad equivalente a las de los más utilizados. El código combinado con todas las optimizaciones es díficil de leer porque consiste en la combinación de todos los detalles técnicos descriptos juntos. Si bien es posible optimizar aun más la memoria utilizando un método in-place que no utilice arreglos adicionales para almacenar datos parciales, como de todos modos es necesario utilizar un arreglo de $N$ elementos para almacenar $W[k]$, dicha técnica no reduciría asintóticamente el uso de memoria adicional. El algoritmo final demoró, en una computadora moderna del 2015 con Windows 10 unos 127ms en calcular 100 FFT's de 4096 puntos. Fue verificado su correcto funcionamiento en comparación con el resultado de librerias de Python que calculaban FFT.