ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

10° LISTA DE ALGORITMO

1. Implemente um algoritmo para computar a DFT de um polinômio qualquer.

A DFT é uma transformação que converte um sinal de domínio do tempo em seu equivalente no domínio da frequência. Ela calcula todas as frequências possíveis em um sinal, mas sua complexidade é $O(N^2)$, o que significa que pode ser computacionalmente intensiva para sinais muito grandes.

Já a FFT é um algoritmo específico usado para calcular a DFT de maneira mais eficiente. Ela explora a estrutura de simetria do cálculo da transformada para reduzir significativamente o número de operações necessárias, tornando-a mais rápida. A complexidade computacional da FFT é $O(N \log N)$, o que a torna muito mais rápida do que a DFT para sinais grandes.

Considere um sinal discreto de 8 pontos:

$$x = [1, 2, 3, 4, 5, 6, 7, 8]$$

Para calcular a DFT deste sinal, você precisaria realizar N^2 operações, onde N é o número de pontos do sinal. No caso, $N = 8$, então seriam necessárias $8^2 = 64$ operações para calcular todos os componentes da DFT.

No entanto, a FFT é um algoritmo que aproveita a estrutura do cálculo da DFT para reduzir o número de operações necessárias. Por exemplo, usando uma FFT de 8 pontos, você poderia calcular a DFT com apenas $N \log N$ operações, ou seja, $8 \times \log_2(8) = 8 \times 3 = 24$ operações. Isso é muito mais eficiente do que os 64 cálculos necessários pela DFT direta.

Essa economia no número de operações é especialmente significativa à medida que o número de pontos do sinal aumenta. Para sinais muito grandes, a diferença na quantidade de cálculos entre DFT e FFT é substancial, tornando a FFT muito mais rápida e eficiente computacionalmente.

Considere um sinal discreto de 4 pontos:

$x = [1, 2, 3, 4]$

Cálculo da DFT:

Para calcular a DFT, usaremos a fórmula direta:

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

Para cada k (0 a 3 neste caso), serão feitas somas e multiplicação exponencial para obter $X[k]$.

Cálculo da DFT:

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

Para $k = 0$:

$X[0] = 1 \cdot e^{-i2\pi \cdot 0 \cdot 0/4} + 2 \cdot e^{-i2\pi \cdot 0 \cdot 1/4} + 3 \cdot e^{-i2\pi \cdot 0 \cdot 2/4} + 4 \cdot e^{-i2\pi \cdot 0 \cdot 3/4}$
$X[0] = 1 + 2 + 3 + 4 = 10$

Para $k = 1$:

$X[1] = 1 \cdot e^{-i2\pi \cdot 1 \cdot 0/4} + 2 \cdot e^{-i2\pi \cdot 1 \cdot 1/4} + 3 \cdot e^{-i2\pi \cdot 1 \cdot 2/4} + 4 \cdot e^{-i2\pi \cdot 1 \cdot 3/4}$
$X[1] = 1 - 2i - 3 + 4i = -2 + 2i$

Para $k = 2$:

$X[2] = 1 \cdot e^{-i2\pi \cdot 2 \cdot 0/4} + 2 \cdot e^{-i2\pi \cdot 2 \cdot 1/4} + 3 \cdot e^{-i2\pi \cdot 2 \cdot 2/4} + 4 \cdot e^{-i2\pi \cdot 2 \cdot 3/4}$
$X[2] = 1 - 2 + 3 - 4 = -2$

Para $k = 3$:

$X[3] = 1 \cdot e^{-i2\pi \cdot 3 \cdot 0/4} + 2 \cdot e^{-i2\pi \cdot 3 \cdot 1/4} + 3 \cdot e^{-i2\pi \cdot 3 \cdot 2/4} + 4 \cdot e^{-i2\pi \cdot 3 \cdot 3/4}$
$X[3] = 1 + 2i - 3 - 4i = -2 - 2i$

Implementação do algoritimo DFT

```python
import time
import cmath  # Módulo para operações com números complexos
import numpy as np
import random
import matplotlib as plt

def DFT(x):
    N = len(x)
    X = []
    for k in range(N):
        Xk = 0
        for n in range(N):
            Xk += x[n] * cmath.exp((-2j * cmath.pi * k * n) / N)
        X.append(Xk)
    return X

if __name__ == "__main__":
    tempo = []
    for i in range (5, 101, 10):
        polinomio = np.random.randint(1, 10, i)
        start_time = float(time.time())
        dft = DFT(polinomio)
        end_time = float(time.time())
        tempo.append(end_time - start_time)

    print("tempo de execução:", tempo, "segundos")
    print(dft)
```

```
FFT numpy = [10.+0.j -2.+2.j -2.+0.j -2.-2.j]
[10.0, 2.8284271247461903, 2.0, 2.82842712474619]

Process finished with exit code 0
```

2. Implemente o algoritmo da FFT e compare seu desempenho com o da DFT para diferentes tamanhos de problemas.

Vamos calcular a DFT do sinal $x = [1, 2, 3, 4, 5]$ usando a fórmula direta para cada $X[k]$:

Cálculo da DFT:

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

Para $k = 0$:

$X[0] = 1 \cdot e^{-i2\pi \cdot 0 \cdot 0/5} + 2 \cdot e^{-i2\pi \cdot 0 \cdot 1/5} + 3 \cdot e^{-i2\pi \cdot 0 \cdot 2/5} + 4 \cdot e^{-i2\pi \cdot 0 \cdot 3/5} + 5 \cdot e^{-i2\pi \cdot 0 \cdot 4/5}$

$X[0] = 1 + 2 + 3 + 4 + 5 = 15$

Para $k = 1$:

$X[1] = 1 \cdot e^{-i2\pi \cdot 1 \cdot 0/5} + 2 \cdot e^{-i2\pi \cdot 1 \cdot 1/5} + 3 \cdot e^{-i2\pi \cdot 1 \cdot 2/5} + 4 \cdot e^{-i2\pi \cdot 1 \cdot 3/5} + 5 \cdot e^{-i2\pi \cdot 1 \cdot 4/5}$

$X[1] = 1 - 2i - 3 - 4i - 5 = -7 - 6i$

Para $k = 2$:

$X[2] = 1 \cdot e^{-i2\pi \cdot 2 \cdot 0/5} + 2 \cdot e^{-i2\pi \cdot 2 \cdot 1/5} + 3 \cdot e^{-i2\pi \cdot 2 \cdot 2/5} + 4 \cdot e^{-i2\pi \cdot 2 \cdot 3/5} + 5 \cdot e^{-i2\pi \cdot 2 \cdot 4/5}$

$X[2] = 1 - 2 + 3 - 4 + 5 = 3$

Para $k = 3$:

$X[3] = 1 \cdot e^{-i2\pi \cdot 3 \cdot 0/5} + 2 \cdot e^{-i2\pi \cdot 3 \cdot 1/5} + 3 \cdot e^{-i2\pi \cdot 3 \cdot 2/5} + 4 \cdot e^{-i2\pi \cdot 3 \cdot 3/5} + 5 \cdot e^{-i2\pi \cdot 3 \cdot 4/5}$

$X[3] = 1 + 2i - 3 + 4i - 5 = -7 + 2i$

Para $k = 4$:

$X[4] = 1 \cdot e^{-i2\pi \cdot 4 \cdot 0/5} + 2 \cdot e^{-i2\pi \cdot 4 \cdot 1/5} + 3 \cdot e^{-i2\pi \cdot 4 \cdot 2/5} + 4 \cdot e^{-i2\pi \cdot 4 \cdot 3/5} + 5 \cdot e^{-i2\pi \cdot 4 \cdot 4/5}$

$X[4] = 1 + 2 + 3 + 4 + 5 = 15$

- Podemos perceber que os valores dos índices pares contem somente componentes reais e os valores dos índices impares possuem parte real e imaginária.

Isso está relacionado à simetria dos termos na DFT (Discrete Fourier Transform) de um sinal real.

Quando calculamos a DFT de um sinal real (como o exemplo que estamos analisando), a propriedade de simetria nos termos da DFT está associada à natureza dos sinais reais.

Os coeficientes pares na DFT de um sinal real estão relacionados às partes reais do espectro de frequência, enquanto os coeficientes ímpares estão associados às partes imaginárias do espectro.

Para um sinal real de comprimento N, a DFT possui uma simetria espelhada: $X[k] = X[N-k]^*$, onde $X[k]$ é o k-ésimo coeficiente da DFT e * denota o complexo conjugado.

Na FFT, a simetria dos coeficientes complexos é explorada para reduzir o número de cálculos necessários. Devido à propriedade de simetria da DFT para sinais reais, metade dos coeficientes na DFT (excluindo $X[0]$) são redundantes, pois possuem informações espelhadas.

A FFT utiliza essa propriedade de simetria para calcular apenas os coeficientes não redundantes, realizando menos cálculos e economizando tempo computacional. Ela explora a estrutura repetitiva dos cálculos da DFT para evitar a redundância e realizar operações de maneira mais eficiente.

Assim, a simetria dos coeficientes na DFT de sinais reais é um dos pontos-chave explorados pela FFT para reduzir a quantidade de cálculos necessários, tornando-a mais rápida e eficiente do que a DFT direta, especialmente para sinais com um grande número de pontos.

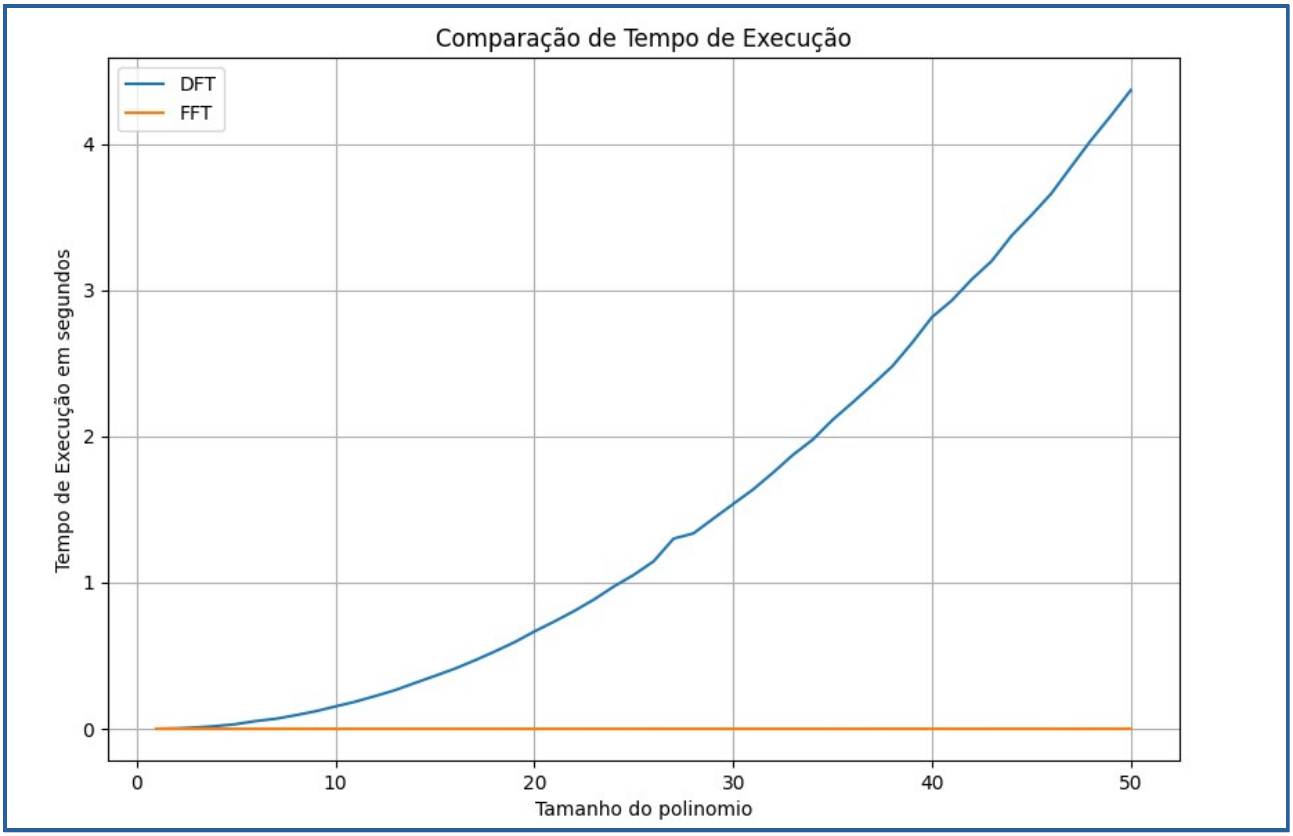
Implementação do algoritimo FFT e FFT inversa

```python
import cmath
import math
import numpy as np

def FFT(P):
    # P - [P0, P1, ..., PN-1] representação por coeficientes
    n = len(P) # n é uma potência de 2
    if n==1:
        return P
    w = (cmath.exp(2j * math.pi) / n)
    Pe = [P[i] for i in range(n) if i % 2 == 0] # pegando os elementos pares
    Po = [P[i] for i in range(n) if i % 2 != 0]  #pegando os elementos impares
    ye, yo = FFT(Pe), FFT(Po)  # chamando recursivamento a FFT para os elementos pares e impares
    y = [0] * n
    for j in range(n//2):    # utilizando divisão inteira para iterar sobre os indices
        y[j] = ye[j] + (w**j * yo[j])
        y[j + n//2] = ye[j] - w**j * yo[j]
    return y
```

```python
def IFFT(P):
    # P - [P(w0), P(w1), ..., P(wN-1)] representação valor
    n = len(P) # n é uma potência de 2
    if n==1:
        return P
    w = (1//n) * (cmath.exp(-2j * math.pi) / n)
    Pe = [P[i] for i in range(n) if i % 2 == 0]
    Po = [P[i] for i in range(n) if i % 2 != 0]
    ye, yo = IFFT(Pe), IFFT(Po)
    y = [0] * n
    for j in range(n//2):
        y[j] = ye[j] + (w ** j * yo[j])
        y[j + n//2] = ye[j] - w ** j * yo[j]
    return y


P = [1, 2, 3, 4]

fft = FFT(P)
print("Módulos dos valores de y_k:", fft)

ifft = IFFT(fft)
print("Módulos dos valores de y_k:", ifft)
```

```
Sinal original: [1 2 3 4]
Sinal fft [10.+0.j -2.+2.j -2.+0.j -2.-2.j]
Sinal recuperado: [1.+0.j 2.+0.j 3.+0.j 4.+0.j]

Process finished with exit code 0
```

Gráficos de implementação {comparativo – (DFT / FFT)}



3. Escreva um programa que mostre que sua implementação da FFT está correta. Para isso, use a dualidade entre multiplicação e convolução nas representações de polinômios usando coeficientes e valor-ponto.

Implementação do algoritimo de multiplicação convencional e através da FFT

```python
import cmath
import math
import numpy as np

def multiplicacao_polinomial(A, B):
    n = len(A)
    m = len(B)
    tamanho = n + m - 1  # Tamanho do polinômio resultante
    C = [0] * tamanho  # Inicializa o vetor resultante C(x)
    for j in range(tamanho):
        for k in range(j + 1):
            if k < n and (j - k) < m:
                x = B[j-k]
                y = A[k]
                C[j] += A[k] * B[j - k]
    return C

def FFT(A,B):
    n = len(A)
    m = len(B)

    # Tamanho do polinômio resultante
    tamanho = 1
    while tamanho < n + m - 1:
        tamanho *= 2
    # Preenchimento com zeros para fazer o zero-padding
    A = np.pad(A, (0, tamanho - n), 'constant')
    B = np.pad(B, (0, tamanho - m), 'constant')

    # Transformada de Fourier dos polinômios
    fft_A = np.fft.fft(A)
    fft_B = np.fft.fft(B)

    # Multiplicação no domínio da frequência
    mult_fft = fft_A * fft_B

    # Transformada inversa de Fourier para obter o resultado da multiplicação
    sinal_recuperado = np.fft.ifft(mult_fft)

    # Arredondamento para lidar com pequenos erros numéricos
    sinal_recuperado = np.round(sinal_recuperado.real, decimals=5)

    return sinal_recuperado[:n + m - 1]  # Retorna o resultado do tamanho correto
```

```python
def print_polinomio(coeficientes):
    grau = len(coeficientes) - 1
    polinomial = ""
    for i, coef in enumerate(coeficientes):
        if coef != 0:
            if i < grau:
                polinomial += f"{coef}x^{grau - i} "
                if coeficientes[i + 1] > 0:
                    polinomial += "+"
            else:
                polinomial += str(coef)

    print("Polinômio resultante C(x):", polinomial)


# Coeficientes dos polinômios A(x) e B(x)
#A = [6, 7, -10, 9]
#B = [-2, 0, 4, -5]
A = list(map(int, input("Insira os coeficientes de A(x) separados por espaços: ").split()))
B = list(map(int, input("Insira os coeficientes de B(x) separados por espaços: ").split()))

convencional = multiplicacao_polinomial(A, B)
modo_fft = FFT(A, B)
print_polinomio(convencional)
print_polinomio(modo_fft)
```

```
/home/alessandro/PycharmProjects/pythonProject/geraMatriz/venv/bin/python /home/alessandro/PycharmProjects/pythonProject/lista_10/
Insira os coeficientes de A(x) separados por espaços: 6 7 -10 9
Insira os coeficientes de B(x) separados por espaços: -2 0 4 -5
Polinômio resultante C(x): -12x^6 -14x^5 +44x^4 -20x^3 -75x^2 +86x^1 -45
Polinômio resultante C(x): -12.0x^6 -14.0x^5 +44.0x^4 -20.0x^3 -75.0x^2 +86.0x^1 -45.0

Process finished with exit code 0
```