# Paradigmas de Solução de Problemas

## Divisão e Conquista – Transformada Rápida de Fourier

Prof. Edson Alves - UnB/FGA

2020

# Sumário

# Transformada Rápida de Fourier

# DFT em $O(N \log N)$

- A divisão e conquista pode ser aplicada no cálculo da DFT para reduzir sua complexidade assintótica
- Na etapa de divisão o sinal é dividido em duas partes de tamanhos aproximadamente iguais
- A conquista acontece quando o sinal tem uma única amostra: neste caso a transformada discreta coincide com a própria amostra
- A fusão permite o cálculo da DFT do sinal a partir das DFTs das duas partes
- Se a fusão for feita em $O(N)$, a recorrência se torna

$$f(N) = 2f(N/2) + (N)$$

- O Teorema Mestre nos diz que a complexidade da transformada passa a ser $O(N \log N)$
- Esta versão da DFT é denominada *Fast Fourier Transform* (FFT)

# Decomposição do sinal FFT

- Considere o sinal $(a_k) = a_0, a_1, \ldots, a_{N-1}$
- Assuma, sem perda de generalidade, que $N = 2^t$, para algum $t$ natural
- Se $N$ não for uma potência de dois, basta adicionar um número suficiente de amostras $a_i = 0$ ao sinal até que $N$ se torne uma potência de dois
- A etapa de divisão, também denominada decomposição do sinal, o sinal é separado em duas partes de tamanho $N/2$: as amostras cujos índices são pares $(e_k)$ e as amostras cujos índices são ímpares $(o_k)$
- Assim,

$$(e_k) = a_0, a_2, a_4, \ldots, a_{N-2}$$

e

$$(o_k) = a_1, a_3, a_5, \ldots, a_{N-1}$$

# Visualização da decomposição do sinal

$$\boxed{a_0 \quad \underline{a_1} \quad a_2 \quad \underline{a_3} \quad a_4 \quad \underline{a_5} \quad a_6 \quad \underline{a_7}}$$

$$\boxed{a_0 \quad \underline{a_2} \quad a_4 \quad \underline{a_6}} \qquad \boxed{a_1 \quad \underline{a_3} \quad a_5 \quad \underline{a_7}}$$

$$\boxed{a_0 \quad \underline{a_4}} \quad \boxed{a_2 \quad \underline{a_6}} \quad \boxed{a_1 \quad \underline{a_5}} \quad \boxed{a_3 \quad \underline{a_7}}$$

$$\boxed{a_0} \; \boxed{a_4} \; \boxed{a_2} \; \boxed{a_6} \; \boxed{a_1} \; \boxed{a_5} \; \boxed{a_3} \; \boxed{a_7}$$

# Decomposição $\times$ ordenação

- Gerando a decomposição por meio da alocação de novos dois subvetores com as cópias dos elementos de índices pares e ímpares permite uma implementação *top-down* da FFT
- Para uma implementação *bottom-up*, é preciso entender o padrão subjacente que surge desta decomposição
- De fato, os elementos que ocupam as folhas nas árvores de decomposição tem índices que correspondem à ordenação dos números $\{0, 1, 2, \ldots, N-1\}$ usando como critério a inversão de sua representação binária
- Assim, por meio de um comparador customizado o este ordenação pode ser feita com complexidade $O(N \log N)$, o que não modifica a complexidade da FFT como um todo

# Visualização da ordenação por padrão binário invertido

| Índice | Padrão invertido | Padrão original |
|---|---|---|
| 0 | 000 | 000 |
| 4 | 001 | 100 |
| 2 | 010 | 010 |
| 6 | 011 | 110 |
| 1 | 001 | 100 |
| 5 | 101 | 101 |
| 3 | 110 | 011 |
| 7 | 111 | 111 |

# Implementação da ordenação por padrão binário

```cpp
#include <bits/stdc++.h>

using namespace std;

int reversed(int x, int bits)
{
    int res = 0;

    for (int i = 0; i < bits; ++i)
    {
        res <<= 1;
        res |= (x & 1);
        x >>= 1;
    }

    return res;
}

```

# Implementação da ordenação por padrão binário

```cpp
template<typename T> vector<T> sortByBits(const vector<T>& xs)
{
    int N = (int) xs.size(), bits = 1;

    while ((1 << bits) != N)
        ++bits;

    vector<int> is(N);
    iota(is.begin(), is.end(), 0);

    sort(is.begin(), is.end(), [&bits](int x, int y) {
        return reversed(x, bits) < reversed(y, bits);
    });

    vector<T> ans(N);

    for (int i = 0; i < N; ++i)
        ans[i] = xs[is[i]];

    return ans;
}
```

# Conquista

- A etapa de conquista acontece em sinais como uma única amostra
- Aplicando o valor $N = 1$ na transformada discreta obtêm-se

$$X_0 = \sum_{n=0}^{N-1} x_n e^{-2\pi ikn/N} = \sum_{n=0}^{0} x_0 e^{-2\pi kn} = x_0$$

- Assim, a própria amostra corresponde à sua transformada
- É preciso atentar, contudo, que embora numericamente iguais, $X_0$ reside no domínio das frequências, enquanto que $x_0$ está no domínio do tempo
- Portanto esta etapa tem complexidade $O(1)$

# Fusão (Síntese)

- A última etapa consiste em combinar as transformadas das duas partes (pares e ímpares) na transformada do sinal
- Lembrando que a Transformada de Fourier é Linear, a transformada de um sinal $x_k$ pode ser computada como a soma de dois sinais distintos cuja soma resulte em $x_k$
- Considere os sinais $e_k$ e $o_k$ dados por

$$e_i = \begin{cases} x_i, & \text{se } i \text{ é par} \\ 0, & \text{caso contrário} \end{cases}$$

e

$$o_j = \begin{cases} x_j, & \text{se } j \text{ é ímpar} \\ 0, & \text{caso contrário} \end{cases}$$

# Fusão (Síntese)

- Deste modo, $x_k = e_k + o_k$
- As transformadas de $e_k$ e $o_k$ tem um comportamento peculiar
- A transformada de $e_k$ duplica seus resultados
- A transformada de $o_k$ tem mesmo comportamento, porém com os valores multiplicados por uma componente sinusoidal
- Isto porque, em relação à $x_k$, o sinal $o_k$ está deslocado no tempo em uma unidade
- Deslocar no tempo corresponde a convolução do sinal com uma função $\delta(t-a)$, onde $a$ é o deslocamento
- A transformada da função $\delta(t-a)$ é uma exponencial complexa:

$$\mathcal{F}[\delta(t-a)] = \int_{-\infty}^{\infty} \delta(t-a)e^{-2\pi kit}dt = e^{-2\pi kai}$$

# Visualização dos sinais no domínio do tempo

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $x_k$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ |
| $e_k$ | $x_0$ | $0$ | $x_2$ | $0$ | $x_4$ | $0$ | $x_6$ | $0$ |
| $o_k$ | $0$ | $x_1$ | $0$ | $x_3$ | $0$ | $x_5$ | $0$ | $x_7$ |

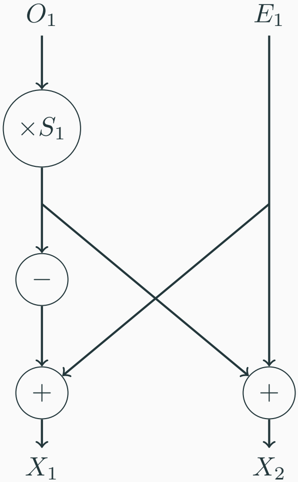
# Visualização das transformadas no domínio das frequências

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $X_k$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $X_8$ |
| $E_k$ | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_1$ | $E_2$ | $E_3$ | $E_4$ |
| $O_k$ | $s_1O_1$ | $s_2O_2$ | $s_3O_3$ | $s_4O_4$ | $s_1O_1$ | $s_2O_2$ | $s_3O_3$ | $s_4O_4$ |

# Padrão borboleta

- Como as transformadas das duas partes foram computadas sem o deslocamento, é preciso compensá-lo na composição da transformada do todo
- As componentes oriundas da parte par $E_k$ são somadas sem alteração em cada componente da transformada $X_k$
- Já as componentes de $O_k$ devem ser multiplicadas pela componente sinusoidal
$$S_k = e^{\frac{2\pi k i}{N}}$$
antes de serem somadas
- Esta multiplicação afeta o sinal do termo: a primeira metade terá sinal negativo, e a segunda metade sinal positivo
- Este ajuste é denominado padrão borboleta, por conta da visualização do diagrama gerado

# Padrão borboleta para $N = 2$



# Implementação

# Implementação recursiva *top-down*

- Uma forma de implementar a FFT é por meio de recursão
- A cada etapa, são criados dois subvetores $e_k$ e $o_k$, de tamanho $N/2$, e a transformada é chamada em cada um destes subvetores
- O caso base acontece quando $N = 1$, onde a transformada é igual à própria amostra
- Na etapa de síntese ou fusão, os coeficientes da transformada $X_k$ podem ser computado atráves das transformadas $E_k$ e $O_k$ dos subvetores:

$$X_j = \begin{cases} E_j + S_j O_j, & \text{se } 0 \leq j < N/2, \\ E_{j-N/2} - S_{j-N/2} O_{j-N/2}, & \text{caso contrário} \end{cases}$$

onde

$$S_j = e^{-2\pi j i}$$

## Implementação recursiva *top-down*

- É possível implementar tanto a transformada quanto a inversa em uma função
- Primeiramente, é preciso uniformizar o tipo dos vetores
- Uma opção é que todos sejam vetores de complexos
- Em segundo lugar, são duas as diferenças entre a transformada e sua inversa:
    1. os sinais dos ângulos são opostos
    2. na inversa os coeficientes são divididos por $N$
- Uma *flag* booleana pode ser passada como parâmetro para decidir o sentido da transformada
- Como $N$ é uma potência de 2, uma forma de realizar esta divisão por $N$ é dividir, a cada etapa, os coeficientes por 2

# Implementação *top-down* da FFT

```
#include <bits/stdc++.h>

using namespace std;

const double PI { acos(-1.0) };

void fft(vector<complex<double>>& xs, bool invert = false)
{
    int N = (int) xs.size();

    if (N == 1)
        return;

    vector<complex<double>> es(N/2), os(N/2);

    for (int i = 0; i < N/2; ++i)
        es[i] = xs[2*i];

    for (int i = 0; i < N/2; ++i)
        os[i] = xs[2*i + 1];

```

# Implementação *top-down* da FFT

```
    fft(es, invert);
    fft(os, invert);

    auto signal = (invert ? 1 : -1);
    auto theta = 2 * signal * PI / N;
    complex<double> S { 1 }, S1 { cos(theta), sin(theta) };

    for (int i = 0; i < N/2; ++i)
    {
        xs[i] = (es[i] + S * os[i]);
        xs[i] /= (invert ? 2 : 1);

        xs[i + N/2] = (es[i] - S * os[i]);
        xs[i + N/2] /= (invert ? 2 : 1);

        S *= S1;
    }
}
```

# Implementação *bottom-up*, *in-place*

- Utilizando a ordenação por *bits* invertidos, é possível implementar a FFT *bottom-up*
- Além de dispensar a recursão, o custo de memória é reduzido, pois só é preciso copiar dois coeficientes a cada atualização, sem precisar copiar os subvetores a cada iteração
- A ordenação baseada em *bits* pode ser feita em $O(N)$: basta trocar de posição dos elementos de índices $i$ e $j$ tais que $i < j$ e que $i$ e $j$ são mutuamente reversos

# Implementação *bottom-up, in-place* da FFT

```cpp
#include <bits/stdc++.h>

using namespace std;

const double PI { acos(-1.0) };

int reversed(int x, int bits)
{
    int res = 0;

    for (int i = 0; i < bits; ++i)
    {
        res <<= 1;
        res |= (x & 1);
        x >>= 1;
    }

    return res;
}

```

# Implementação *bottom-up, in-place* da FFT

```
void fft(vector<complex<double>>& xs, bool invert = false)
{
    int N = (int) xs.size();

    if (N == 1)
        return;

    int bits = 1;

    while ((1 << bits) != N)
        ++bits;

    for (int i = 0; i < N; ++i)
    {
        auto j = reversed(i, bits);

        if (i < j)
            swap(xs[i], xs[j]);
    }

```

# Implementação *bottom-up, in-place* da FFT

```cpp
    for (int size = 2; size <= N; size *= 2)
    {
        auto signal = (invert ? 1 : -1);
        auto theta = 2 * signal * PI / size;
        complex<double> S1 { cos(theta), sin(theta) };

        for (int i = 0; i < N; i += size)
        {
            complex<double> S { 1 }, k { invert ? 2.0 : 1.0 };

            for (int j = 0; j < size / 2; ++j)
            {
                auto a { xs[i + j] }, b { xs[i + j + size/2] * S };

                xs[i + j] = (a + b) / k;
                xs[i + j + size/2] = (a - b) / k;
                S *= S1;
            }
        }
    }
}
```

# Referências

## Referências


1. **CHEEVER**, Erick. The Fourier Series, acesso em 12/08/2020.
2. CP Algorithms. Fast Fourier Transform, acesso em 13/08/2020.
3. **SMITH**, Steven W. The Scientist and Engineer's Guide to Digital Signal Processing, acesso em 17/08/2020.
4. Standford. Lecture 11 – The Fourier Transform, acesso em 13/08/2020.
5. The Fourier Transform.com. The Dirac-Delta Function – The Impulse, acesso em 18/08/2020.
6. Wikipédia. Discrete Fourier Transform, acesso em 13/08/2020.
7. Wolfram. Fourier Series, acesso em 12/08/2020.
8. Wolfram. Fourier Transform, acesso em 13/08/2020.