Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Luis Gustavo Bitencourt Almeida

Transformada Rápida de Fourier em Programação Competitiva

São Paulo
Novembro de 2018

Transformada Rápida de Fourier em Programação Competitiva

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: José Coelho de Pina Junior

São Paulo
Novembro de 2018

Agradecimentos

Agradeço ao meu orientador pelo tempo dedicado em me ajudar a escrever este texto, pelos ensinamentos dados em todas as reuniões e pelo café de toda reunião. Ao Marcos Kawakami pela sugestão de tema.

Não posso deixar de agradecer minha mãe. Costureira guerreira me criou sozinha e permitiu que mesmo dentro da realidade das periferias eu pudesse sonhar além do que nos era oferecido.

Agradeço a Marcela pela enorme paciência, dedicação e amor nesses anos ocupados. Um abraço confortável capaz de acalmar tempos difíceis.

Um agradecimento mais do que especial para o MaratonIME inteiro. O grupo é formado por pessoas incríveis que têm elevado o nome do curso de Ciência da Computação do IME. Gostaria de destacar o trabalho maravilhoso do Renzo em fazer todo mundo estudar e treinar para as competições. A maratona de programação me deu muito e espero que com este trabalho possa retribuir um pouco ou pelo menos mostrar minha enorme gratidão.

Resumo

Polinômios são ferramentas poderosas para modelar diversos problemas. É necessário que as operações sejam feitas de maneira eficiente. A transformada rápida de Fourier resolve o problema da multiplicação de polinômios e é útil na aplicação de filtros em imagens e áudios, decomposição de sinais e mudança de representação. Erros de arredondamento fazem necessário o uso de aritmética modular quando possível. Este texto apresenta soluções para valoração, adição e multiplicação de polinômios e mostra alguns usos da transformada de Fourier na programação competitiva.

Palavras-chave: polinômios, fft, convolução, Fourier, competição.

Conteúdo

Capítulo 1

Introdução

Jean-Baptiste Joseph Fourier foi um matemático e físico francês que viveu entre 1768 e 1830. O nome de Fourier figura entre os setenta e dois nomes de cientistas e engenheiros franceses gravados na Torre Eiffel em reconhecimento às suas contribuições à ciência [10].

Fourier é muito conhecido pelo seu estudo de séries, as hoje chamadas *séries de Fourier*, e suas aplicações em modelar transferência de calor [5] e vibrações. Seu trabalho sobre a transferência de calor resultou no que hoje é conhecido como *transformada de Fourier*.

Acreditava-se que computar a transformada de n pontos (complexos ou reais) consumia tempo proporcional à n^2 até que Cooley e Tukey [1] propuseram um algoritmo em 1965 que ficou conhecido como a *transformada rápida de Fourier* e consome tempo $\mathrm{O}(n \lg n)$ dando luz ao tópico. Vale ressaltar que as primeiras aparições da transformada rápida de Fourier foram posteriormente atribuídas a Gauss em seus trabalhos com cálculos em astronomia [7] para interpolar a órbita de asteroides com amostras igualmente espaçadas [2].

Em geral a transformada de Fourier pode ser usada em duas situações. A primeira é para investigar frequências e energia de imagens. A segunda é como sub-rotina de um algoritmo em um *filtro* calculando-se uma *convolução*. Nesse caso o objetivo não é interpretar o resultado da transformada, mas sim usar a transformada rápida de Fourier (FFT) para mover rapidamente entre representações (tempo e frequência). Isso é particularmente útil devido ao teorema da convolução que permite que muitas operações em um domínio sejam feitas no outro de maneira muito eficiente.

Essa mudança de representação de forma eficiente tem sido abordada em problemas de *programação competitiva*. Especialmente, uma variação da transformada de Fourier que usa valores igualmente espaçados conhecida como *transformada discreta de Fourier*. Problemas como multiplicação de inteiros e multiplicação de polinômios aparecem com frequência em problemas de contagem, geometria e etc. Ainda que polinômios não apresentem relação com a transformada de Fourier imediatamente, serão fundamentais na apresentação das ideias.

Programação competitiva é uma modalidade de competição em que times ou indivíduos recebem uma lista de problemas computacionais que devem ser resolvidos em um certo tempo, tipicamente em algumas horas. As soluções desses problemas consistem de programas que devem ser submetidos a avaliação automática feita por um outro programa, o chamado *juiz eletrônico*. Cada solução submetida deve resolver uma série de casos de teste. As respostas do programa serão comparadas pelo juiz eletrônico com um gabarito. As séries de testes devem ser resolvidas obedecendo limites de memória e de tempo. Esses limites procuram garantir que a solução atenda a exigências de eficiência. Aqui a eficiência da FFT entra em ação para resolver alguns dos problemas.

Este texto está organizado da seguinte maneira. No capítulo 2, tratamos de polinômios, descrevemos algumas de suas representações e estudamos as operações sobre polinômios, em

particular a operação de multiplicação de polinômios que está intimamente relacionada a convolução. O consumo de tempo de cada operação depende fortemente da representação. Em seguida, no capítulo 3, introduzimos a FFT. A FFT é o algoritmo mais eficiente conhecido para transitarmos entre representações de polinômios. Essa transição é o mecanismo chave que permite que realizemos a operação de multiplicação de polinômios no contexto em que ela é mais eficiente. Algumas aplicações da FFT em programação competitiva são mostradas no capítulo 4. Finalmente, no capítulo 5, apresentamos as nossas conclusões.

Capítulo 2

Polinômios

Um **polinômio** é uma expressão formada por coeficientes e variáveis envolvendo somente as operações de adição, subtração, multiplicação e exponenciação das variáveis por inteiros não negativos. Neste texto trataremos de polinômios com apenas uma variável. Isto é, um polinômio é uma função da forma

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} = \sum_{k=0}^{n-1} a_k x^k$$

O grau de um polinômio é o maior expoente de suas variáveis. Os valores a_k são os **coeficientes do polinômio**, em geral são valores reais. A figura 2.1 mostra um exemplo de gráfico de um polinômio de grau 4.

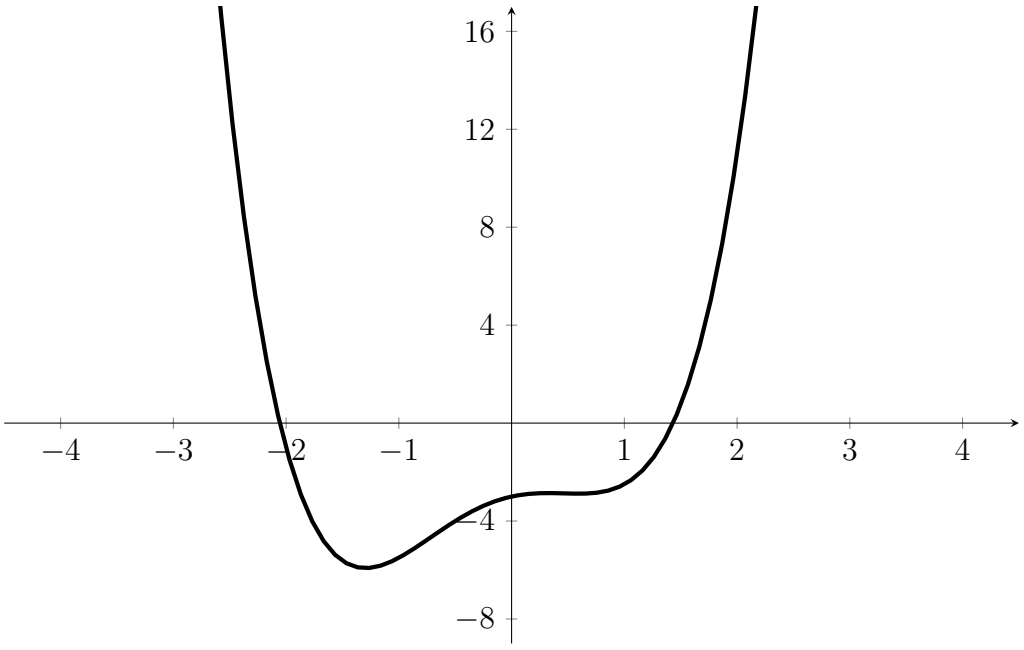

Figura 2.1: *Gráfico do polinômio* $A(x) = x^4 + 0.5x^3 - 2x^2 + x - 3$.

Polinômios são usados em vários contextos para representar objetos e modelar fenômenos. Em cálculo e análise, polinômios são usados para aproximar funções mais complexas [13]. Em geometria, polinômios codificam variedades algébricas [8]. Na próxima seção abordaremos formas de representar e fazer operações em polinômios.

2.1 Representações

Nesta seção cobriremos duas formas de representar um polinômio. A primeira e mais usual é a representação por coeficientes como descrito anteriormente. Para os algoritmos no texto, um polinômio $A(x)$ no formato de coeficientes será representado por um **vetor de coeficientes** a, onde a_k corresponde ao coeficiente de $a_k x^k$.

A segunda representação é a representação por **pontos** (*point-value representation*). As representações são equivalentes, de certa forma. Embora hajam infinitas representações por pontos, é possível recuperar os coeficientes com o processo de **interpolação** dos pontos. O teorema a seguir mostra que a interpolação de n pontos é bem definida.

Teorema 1 (Unicidade de um polinômio interpolador)**.** *Seja $A(x)$ um polinômio de grau $n-1$. Os coeficientes de $A(x)$ são determinados de maneira única a partir da lista de pontos $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ tal que $y_k = A(x_k)$ para $k = 0, 1, \ldots, n-1$ e $x_i \neq x_j$.*

Demonstração. Considere o seguinte sistema de equações

$$\begin{array}{ccccccccccc}
a_0 & + & a_1 x_0 & + & a_2 x_0^2 & + & \ldots & + & a_{n-1} x_0^{n-1} & = & y_0 \\
a_0 & + & a_1 x_1 & + & a_2 x_1^2 & + & \ldots & + & a_{n-1} x_1^{n-1} & = & y_1 \\
\vdots & & & & & & \ddots & & & & \\
a_0 & + & a_1 x_{n-1} & + & a_2 x_{n-1}^2 & + & \ldots & + & a_{n-1} x_{n-1}^{n-1} & = & y_{n-1}
\end{array}$$

Em representação matricial esse sistema pode ser escrito como

$$\begin{bmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\
1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\
1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1}
\end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} \tag{2.1}$$

A matriz à esquerda é conhecida como **matriz de Vandermonde**. Ela é denotada por $V(x_0, x_1, \ldots, x_{n-1})$ e tem determinante

$$\prod_{0 \leq j < k \leq n-1} (x_k - x_j)$$

Como todos os pontos x_k são distintos, esse determinante é diferente de zero. Logo, a matrizes de Vandermonde são não singulares e portanto inversíveis. Assim, os coeficientes de $A(x)$ podem ser recuperados unicamente por

$$\begin{bmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\
1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\
1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1}
\end{bmatrix}^{-1} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} \tag{2.2}$$

■

Veremos na seção 2.2 como ambas as representações se comportam ao calcular um novo valor de $A(x)$ e ao somar e multiplicar dois polinômios $A(x)$ e $B(x)$.

2.2 Operações

Existem pelo menos três operações elementares envolvendo polinômios. São elas: valoração, adição e multiplicação. A primeira associa a um polinômio e um valor tipicamente real ou complexo, um outro valor real ou complexo. As outras duas associam dois polinômios a um terceiro polinômio.

Veremos a seguir como cada operação é feita em cada representação de um polinômio. Ao fim iremos comparar as operações por representação e analisar maneiras eficientes de computar cada uma delas.

Valoração

Dado um polinômio $A(x)$ queremos calcular o valor de $A(x)$ para um valor específico de x, digamos x_0.

Na representação de coeficientes o polinômio $A(x)$ é dado pelo vetor $a = \langle a_0, a_1, \ldots, a_{n-1}\rangle$. Uma maneira natural de obtermos o valor de $A(x_0)$ é computarmos x_0^k e multiplicar por a_k.

```
VALORAÇÃO(a, x)
1  value = 0
2  for k = 0 to n − 1
3        x_k = 1
4        for j = 1 to k
5              x_k = x_k × x
6        value = value + a_k × x_k
7  return value
```

O consumo de tempo desse cálculo é dado pela recorrência

$$T(n) = T(n-1) + \Theta(n) = \frac{n(n-1)}{2} = \Theta(n^2)$$

onde $T(n)$ é o número de execuções da linha *5*.

Uma outra maneira de escrever o polinômio $A(x)$ é

$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1}))) \tag{2.3}$$

A expressão (2.3) sugere uma forma para calcular $A(x_0)$ em tempo linear conhecido como **método de Horner**, descrito abaixo.

Note que a_{n-1} é multiplicado $n-1$ vezes por x. Em termos gerais, a_k é multiplicado k vezes por x e, na iteração decrescente em j, x multiplica todos os coeficientes maiores ou iguais a j. Assim, basta manter o resultado até a iteração j armazenado, multiplicar por x e então somar a_j.

A função recebe um polinômio $A(x)$ no formato de vetor de coeficientes a e um ponto x e devolve o valor de $A(x)$.

Note que a linha *4* é executada uma vez para cada coeficiente. Portanto, o consumo de tempo $T(n)$ da função HORNER(a, x) é $\Theta(n)$.

HORNER(a, x)
1 $value = 1$
2 $k = n - 1$
3 **while** $k \geq 0$
4 $\quad value = value \times x + a_k$
5 $\quad k = k - 1$
6 **return** $value$

Já na representação por pontos não é possível calcular um novo valor $A(x_0)$. É necessário interpolar o polinômio a partir dos pontos e calcular esse valor no formato de coeficientes. O método sugerido na equação (2.2) exige que computemos o inverso da matriz de Vandermonde. O método mais clássico para calcular o inverso de uma matriz qualquer é o de **eliminação Gaussiana**, que consome tempo $O(n^3)$.

Um algoritmo mais rápido para interpolar um polinômio é baseado na **fórmula de Lagrange** [12].

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)} \tag{2.4}$$

Portanto, calcular um novo valor de $A(x)$ na representação por pontos pode ser feita em tempo $\Theta(n^2)$.

Adição

A **adição** de dois polinômios é um novo polinômio $C(x)$ tal que $C(x) = A(x) + B(x)$ para todo x.

Na representação por coeficientes, a adição de dois polinômios é feita somando os coeficientes de mesmo índice. Assim, se $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ e $b = \langle b_0, b_1, \ldots, b_{n-1} \rangle$ são dois polinômios de grau $n - 1$ a sua soma $C(x)$ é o polinômio

$$C(x) = A(x) + B(x) = \sum_{k=0}^{n-1} (a_k + b_k) x^k \tag{2.5}$$

Se o grau dos polinômios é diferente os coeficientes basta colocar zero nos coeficientes que faltam.

A seguir ilustramos esse processo adicionando os polinômios $x^3 + 4x^2 + 3$ e $7x^2 + x + 3$.

$$\begin{array}{rrrrrrrrr} & x^3 & + & 4x^2 & + & & + & 3 & \langle 3, 0, 4, 1 \rangle \\ + & & + & 7x^2 & + & x & + & 3 & \langle 3, 1, 7, 0 \rangle \\ \hline & x^3 & + & 11x^2 & + & x & + & 6 & \langle 6, 1, 11, 1 \rangle \end{array}$$

Na representação de pontos, por definição, basta somar os valores ponto a ponto.

Por (2.5) temos que a soma de dois polinômios na representação de coeficientes pode ser determinada fazendo-se $n - 1$ adições. Ou seja, em ambas as representação a soma de dois polinômios pode ser obtida em tempo $\Theta(n)$.

Multiplicação

Dados dois polinômios $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ e $B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1}$, queremos computar um polinômio $C(x) = A(x) \cdot B(x)$ para todo x. O termo c_k do polinômio $C(x)$ pode ser escrito como

$$c_k = \sum_{j=0}^{k} a_j b_{k-j} \text{ para } k = 0, 1, \ldots, 2(n-1) \tag{2.6}$$

A equação (2.6) é dita **convolução** dos vetores a e b. O termo convolução será muito utilizado na resolução de problemas no capítulo 4.

A seguir a ilustração extraída do CLRS [4] ilustra a multiplicação dos polinômios $6x^3 + 7x^2 - 10x + 9$ e $-2x^3 + 4x - 5$.

$$\begin{array}{rrrrrrrrrrr}
 & & & & & 6x^3 & + & 7x^2 & - & 10x & + & 9 \\
 & & & & - & 2x^3 & + & & + & 4x & - & 5 \\
\hline
 & & & & - & 30x^3 & - & 35x^2 & + & 50x & - & 45 \\
 & & & 24x^4 & + & 28x^3 & - & 40x^2 & + & 36x & & \\
- & 12x^6 & - & 14x^5 & + & 20x^4 & - & 18x^3 & & & & \\
\hline
- & 12x^6 & - & 14x^5 & + & 44x^4 & - & 20x^3 & - & 75x^2 & + & 86x & - & 45
\end{array}$$

Assim, vemos que multiplicar dois polinômios na representação de coeficientes consome tempo $\Theta(n^2)$, pois para cada coeficiente a_k de $A(x)$ é necessário multiplicar a_k por todos os coeficientes de $B(x)$.

Já na representação por pontos, assim como na adição, a multiplicação é feita como na definição da operação. O polinômio $C(x)$ é obtido fazendo o produto ponto por ponto. Como o grau de $C(x)$ é $2(n-1)$ é preciso ter $2n$ pontos para computar C. Portanto, o consumo de tempo é $\Theta(n)$.

A tabela a seguir resume o consumo de tempo para realizarmos as operações sobre polinômios nas duas representações.

	Representação	
Operação	coeficientes	pontos
Valoração	$O(n)$	$O(n^2)$
Adição	$O(n)$	$O(n)$
Multiplicação	$O(n^2)$	$O(n)$

Uma vez que multiplicar dois polinômios em representação de pontos é tão mais eficiente, uma pergunta pertinente é: por que não converter o polinômio na representação de coeficientes para a representação por pontos e então multiplicá-los? Basta calcular o valor dos polinômios em n pontos distintos.

Entretanto, calcular o valor de um polinômio de grau $n-1$ em n pontos é equivalente ao seguinte produto de matrizes

$$\begin{bmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\
1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\
1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\
\vdots & & \ddots & & \vdots \\
1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1}
\end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} \tag{2.7}$$

Sendo assim, amostrar n pontos em um polinômio ingenuamente consome tempo $O(n^2)$.

No capítulo 3 descreveremos um algoritmo eficiente para calcular o valor de um polinômio em um conjunto de n pontos. Entretanto, faremos uso de uma escolha criteriosa de pontos com propriedades necessárias para esse ganho de desempenho. Veremos na seção 3.1 como escolher tais pontos e por que são tão apropriados para nossas operações.

Capítulo 3

Transformada Rápida de Fourier

No capítulo anterior vimos como o método ingênuo para multiplicar dois polinômios é ineficiente. Entretanto, ao mudar a representação, temos um grande ganho de desempenho.

Vamos analisar melhor a ideia de transformar o polinômio da representação de coeficientes para a de pontos.

Considere o polinômio $A(x)$ de grau $n-1$ com n sendo uma potência de dois. Seja

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \dots a_{n-2}x^{n/2-1}$$

o polinômio definido a partir dos coeficientes pares de $A(x)$. De maneira análoga, seja

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

o polinômio definido pelos coeficientes ímpares. Assim, $A^{[0]}(x)$ e $A^{[1]}(x)$ têm metade do grau de $A(x)$. Deste modo, $A(x)$ pode ser escrito como

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{3.1}$$

Vale notar que em ambos os polinômios o monômio associado ao k-ésimo coeficiente é x^k. Considere o seguinte exemplo para um polinômio de grau três:

$$\begin{aligned} a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 &= \underbrace{a_0(x^2)^0 + a_2(x^2)^1}_{A^{[0]}} + \underbrace{x(a_1(x^2)^0 + a_3(x^2)^1)}_{x\cdot A^{[1]}} \\ &= \underbrace{a_0(x^4)^0}_{(A^{[0]})^{[0]}} + \underbrace{x^2(a_2(x^4)^0)}_{x\cdot(A^{[0]})^{[1]}} + x[\underbrace{a_1(x^4)^0}_{(A^{[1]})^{[0]}} + \underbrace{x^2(a_3(x^4)^0)}_{x\cdot(A^{[1]})^{[1]}}] \\ &= a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 \ . \end{aligned}$$

Vamos analisar um algoritmo de divisão e conquista que calcula o valor de $A(x)$ para todo ponto x em um conjunto X de n pontos distintos.

Um algoritmo de divisão e conquista é tipicamente recursivo e em cada nível da recursão tem três passos elementares: divisão, conquista e combinação. No nosso caso, vamos receber um polinômio $A(x)$ e um conjunto X com n pontos e devolver o valor de $A(x_k)$ para todo x_k em X.

Primeiro, se o polinômio tem grau zero, o valor do polinômio $A(x)$ é a_0 para todo x. Portanto, a resposta na base da recursão é a_k para cada x_k. No passo da divisão vamos fatorar o polinômio $A(x)$ nos polinômios $A^{[0]}(x)$ e $A^{[1]}(x)$ descritos acima e elevar todos os pontos de X ao quadrado. Calcule recursivamente os valores de $A^{[0]}(x)$ e $A^{[1]}(x)$ no novo

conjunto, digamos X^2 e combine as respostas com a equação (3.1).

A figura 3.1 exemplifica a árvore de recursão da função TRANSFORMA-RECURSIVO para um polinômio de grau 3.

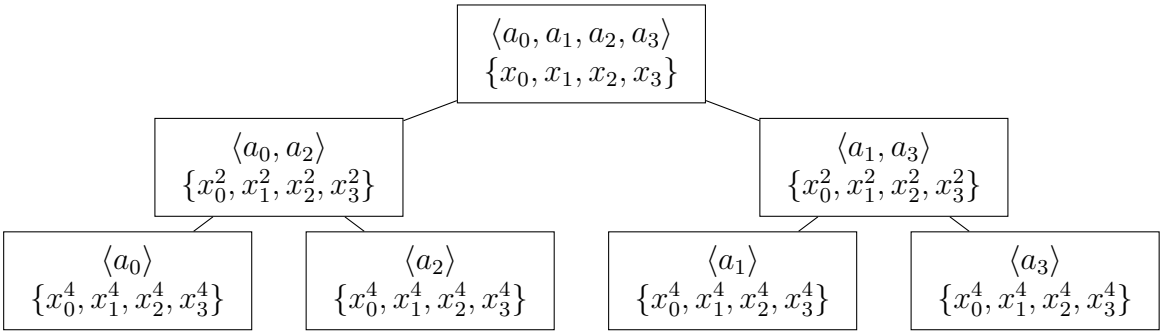

Figura 3.1: *Árvore de recursão para um polinômio de grau 3.*

Após examinar a árvore da recursão, podemos questionar a eficiência deste algoritmo. Embora o tamanho do polinômio de cada nível da árvore tenha metade do tamanho do polinômio anterior, o conjunto X se mantém do mesmo tamanho. A desconfiança é válida: o algoritmo consome tempo $\mathrm{O}(n^2)$ e é na prática ainda mais lento que o método apresentado no capítulo 2.

Note que o tamanho do polinômio diminui a cada iteração. O gargalo de desempenho está no conjunto X. Se o conjunto X fosse tal que $X^2 = \{x^2 : x \in X\}$ tivesse poucos elementos, isso ajudaria no processo.

Diremos que um conjunto X é **colapsante** se $|X| = 1$ ou $|X^2| = |X|/2$ e X^2 é colapsante. Mais precisamente, se fosse possível diminuir X pela metade, ou seja, se X fosse um conjunto colapsante, o algoritmo teria consumo de tempo $\mathrm{O}(n \lg n)$. Na próxima seção vamos estudar um conjunto de pontos que apresenta essa característica de ser colapsante.

TRANSFORMA-RECURSIVO(A, X)

```
1  n = a.length
2  if n == 1
3        return a
4  a^[0] = ⟨a_0, a_2, ..., a_{n-2}⟩
5  a^[1] = ⟨a_1, a_3, ..., a_{n-1}⟩
6  y^[0] = TRANSFORMA-RECURSIVO(a^[0], X^2)
7  y^[1] = TRANSFORMA-RECURSIVO(a^[1], X^2)
8  for k = 0 to n − 1
9        y_k = y_k^[0] + x_k y_k^[1]
10 return y
```

3.1 Raízes da unidade

Estamos interessados em um conjunto de pontos que ao ser elevado ao quadrado resulte em um conjunto no qual metade dos elementos se repitam. O conjunto $X = \{-2,\ 2, -1,\ 1\}$, por exemplo, ao ser elevado ao quadrado se torna $X^2 = \{4, 1\}$. Entretanto X^k possui o mesmo tamanho de X^2 para todo k maior que 2. Portanto, X não é um conjunto colapsante.

Queremos um conjunto de pontos que ao fim da recursão seja um conjunto unitário. Isto é, um conjunto de n pontos x_k tal que $x_k^n - 1 = 0$. As raízes dessa função podem não estar todas no conjunto dos reais, mas com certeza estão no conjunto dos números complexos. Por exemplo, para $X = \{-i,\ i, -1,\ 1\}$ temos que $x^4 = 1$. Aqui i é a unidade imaginária tal que $i = \sqrt{-1}$.

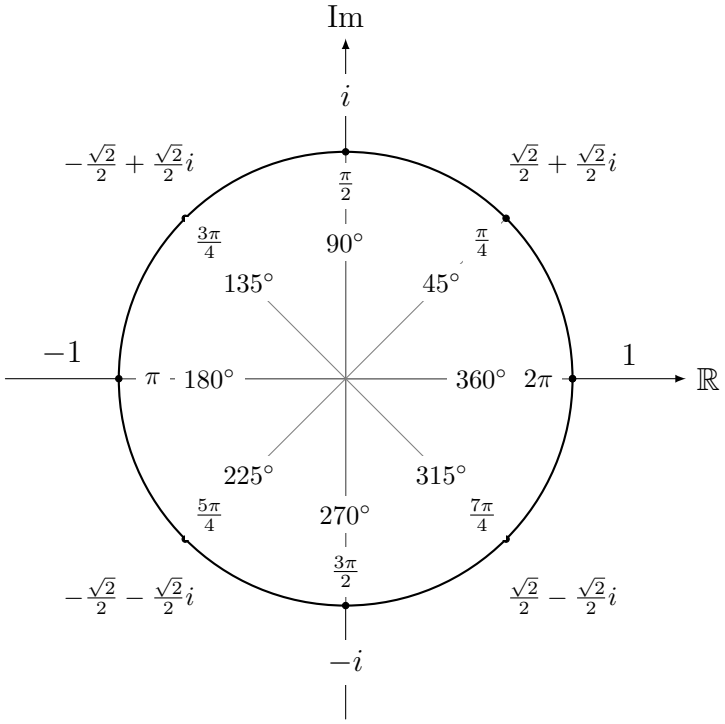

Figura 3.2: *Círculo unitário do plano complexo e as raízes oitavas da unidade.*

Teorema 2. *Para qualquer inteiro positivo n, existem n números complexos ω que satisfazem a equação $\omega^n = 1$. Esses números são as chamadas* ***enésimas raízes complexa da unidade****.*

Considere dois números complexos $\omega_1 = \rho_1 e^{i\theta_1}$ e $\omega_2 = \rho_2 e^{i\theta_2}$. O produto $\omega_1\omega_2$ é igual a $\rho_1\rho_2 e^{(\omega_1+\omega_2)}$. Geometricamente falando, multiplicamos as magnitudes e somamos os ângulos. Se ω_k está no circulo unitário, $\rho_k = 1$ e, portanto, elevar ω_k ao quadrado equivale a dobrar o seu ângulo.

Como exemplificado na figura 3.2, vemos que as raízes da unidade estão localizadas no círculo unitário do plano complexo.

As n raízes complexas da unidade são definidas por $\omega_n^k = e^{2\pi ik/n}$ para $k = 0, 1, \ldots, n-1$. A raiz $\omega_n = e^{2\pi i/n}$ é dita principal enésima raiz da unidade.

Uma propriedade importante desse conjunto é a chamada *Lei do Cancelamento*.

Lema 1 (Lei do Cancelamento)**.** *Para quaisquer números inteiros positivos n, k e d, vale que $\omega_{dn}^{dk} = \omega_n^k$.*

Demonstração. Temos que

$$\begin{aligned}\omega_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^{k} \\ &= \omega_n^k\end{aligned}$$

■

Outra propriedade importante para o nosso estudo é o *Lema de Halving.*

Lema 2 (Lema de Halving)**.** *Se n é um número inteiro positivo par, então os quadrados das enésimas raízes complexas da unidade são as $n/2$-ésimas raízes complexas da unidade.*

Demonstração. Pela Lei do Cancelamento temos que $(\omega_n^k)^2 = \omega_{n/2}^k$, para qualquer k inteiro não negativo. Note que

$$\begin{aligned}(\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2 = \omega_{n/2}^k\end{aligned}$$

Assim, ω_n^k e $\omega_n^{k+n/2}$ possuem o mesmo quadrado e, se elevarmos todas as enésimas raízes da unidade ao quadrado, obtemos as raízes $n/2$-ésimas da unidade exatamente duas vezes. ■

O Lema de Halving é fundamental para que nosso algoritmo de divisão e conquista converta um polinômio da representação por coeficientes para a representação por pontos de maneira eficiente. Isto é, se n é uma potência de 2, o conjunto das n raízes n-ésimas da unidade é um conjunto colapsante.

Lema 3. *Se ω_n é uma enésima raiz da unidade $w_n^{n/2} = -1$.*

Demonstração.

$$\begin{aligned}w_n^{n/2} &= (e^{2\pi i/n})^{n/2} \\ &= e^{\pi i} \\ &= -1\end{aligned}$$

■

3.2 Transformada discreta

Queremos calcular o polinômio $A(x)$ nas n raízes n-ésimas da unidade $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. O vetor $y = (A(\omega_n^0), A(\omega_n^1), \ldots, A(\omega_n^{n-1}))$ é dito **DFT** (*Discrete Fourier Transform*) do polinômio $A(x)$. Vejamos como calcular o vetor y em tempo $\mathrm{O}(n \lg n)$ utilizando o algoritmo Transforma-Recursivo fazendo uso das propriedades das raízes da unidade mostradas na seção 3.1.

Lembrando que na fase de divisão o algoritmo reescreve $A(x)$ como a adição de dois polinômios $A^{[0]}(x)$ e $A^{[1]}(x)$ cada um de grau $n/2 - 1$ da seguinte maneira

$$\begin{aligned}A^{[0]}(x) &= a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}, \\ A^{[1]}(x) &= a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}\end{aligned}$$

e combina a resposta desses polinômios menores com

$$A(x) = A^{[0]}(x^2) + xA^{[1]}. \tag{3.2}$$

Da equação (3.2) temos que o cálculo de $A(x)$ em $X = \{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$ se reduz ao cálculo de $A^{[0]}(x)$ e $A^{[1]}(x)$ em $X^2 = \{(\omega_n^0)^2, (\omega_n)^2, \ldots, (\omega_n^{n-1})^2\}$. Pelo Lema de Halving, esse conjunto X^2 tem metade dos pontos do conjunto original X.

O algoritmo DFT-Recursivo a seguir mostra como calcular o DFT de um polinômio $A(x)$ em representação de coeficientes. O grau do polinômio $A(x)$ é $n-1$ e por conveniência estamos supondo que n é uma potência de dois. A função recebe um vetor de coeficientes a e retorna o DFT de a, ou seja, o vetor y tal que y_k é $A(\omega_n^k)$.

```
DFT-Recursivo(a)
 1  n = a.length
 2  if n == 1
 3        return a
 4  ω_n = e^{2πi/n}
 5  ω = 1
 6  a^[0] = ⟨a_0, a_2, ..., a_{n-2}⟩
 7  a^[1] = ⟨a_1, a_3, ..., a_{n-1}⟩
 8  y^[0] = DFT-Recursivo(a^[0])
 9  y^[1] = DFT-Recursivo(a^[1])
10  for k = 0 to n/2 − 1
11        y_k = y_k^[0] + ω y_k^[1]
12        y_{k+n/2} = y_k^[0] − ω y_k^[1]
13        ω = ω × ω_n
14  return y
```

Vale notar que os vetores $y^{[0]}$ e $y^{[1]}$ possuem metade do tamanho do vetor y. Portanto, é necessário obter os valores de y_k para $k = n/2, n/2+1, \ldots, n-1$. A linha 12 se utiliza do fato mostrado no lema 3 pois

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

O consumo de tempos das linhas 6 e 7 e o bloco 10-13 é $\Theta(n)$. As linhas 8 e 9 são instâncias do mesmo problema com tamanho $n/2$. Portanto, se $T(n)$ é o consumo de tempo da função DFT-Recursivo

$$T(n) = 2T(n/2) + \mathrm{O}(n) = \mathrm{O}(n \lg n)$$

Na figura 3.3 vemos como tanto o polinômio quanto o conjunto de pontos diminuem conforme aumenta a altura da árvore.

Em 1969 Cooley, Lewis e Welch fizeram a comparação entre o método convencional e o método que utiliza as raízes da unidade para calcular o DFT e apresentaram um artigo com os resultados que podem ser vistos em [3].

3.3 Inversa da transformada discreta

No momento sabemos converter um polinômio $A(x)$ em um vetor de pontos A^* tal que $A^*_k = A(x_k)$. Note que no problema de multiplicação de dois polinômios era necessário obter o polinômio $C(x)$ na representação de coeficientes. Como obter o vetor de coeficientes a partir do DFT? Vamos voltar à representação matricial do problema.

Calcular o polinômio $A(x)$ nas enésimas raízes da unidade é equivalente ao produto

$$\begin{bmatrix} 1 & \omega_n^0 & (\omega_n^0)^2 & \dots & (\omega_n^0)^{n-1} \\ 1 & \omega_n^1 & (\omega_n^1)^2 & \dots & (\omega_n^1)^{n-1} \\ 1 & \omega_n^2 & (\omega_n^2)^2 & \dots & (\omega_n^2)^{n-1} \\ \vdots & & \ddots & & \vdots \\ 1 & \omega_n^{n-1} & (\omega_n^{n-1})^2 & \dots & (\omega_n^{n-1})^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} \tag{3.3}$$

A matriz à esquerda é a matriz de Vandermonde $V(\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1})$, e, como dito em (2.1), essa matriz é não singular e portanto inversível. O vetor de coeficientes de C é dado por

$$\begin{bmatrix} 1 & \omega_n^0 & (\omega_n^0)^2 & \dots & (\omega_n^0)^{n-1} \\ 1 & \omega_n^1 & (\omega_n^1)^2 & \dots & (\omega_n^1)^{n-1} \\ 1 & \omega_n^2 & (\omega_n^2)^2 & \dots & (\omega_n^2)^{n-1} \\ \vdots & & \ddots & & \vdots \\ 1 & \omega_n^{n-1} & (\omega_n^{n-1})^2 & \dots & (\omega_n^{n-1})^{n-1} \end{bmatrix}^{-1} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ c_{n-1} \end{bmatrix} \tag{3.4}$$

Portanto, dado o DFT de um polinômio, para recuperar o vetor de coeficientes basta calcular a matriz V^{-1}. Embora o melhor algoritmo para inverter uma matriz qualquer seja o de *Eliminação Gaussiana*, de complexidade $\mathrm{O}(n^3)$, essa matriz possui propriedades interessantes que vão facilitar esse cálculo.

Teorema 3. *Seja V a matriz de Vandermonde $V(\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1})$. $V^{-1} = \frac{1}{n}\overline{V}$, onde $\overline{V}$ é a matriz conjugada complexa de V.*

Demonstração. Nós afirmamos que $P = V \cdot \overline{V} = nI$. De fato,

$$\begin{aligned} p_{jk} &= (\text{linha } j \text{ de } V) \cdot (\text{coluna } k \text{ de } \overline{V}) \\ &= \sum_{m=0}^{n-1} e^{2\pi ijm/n} i \cdot \overline{e^{2\pi ikm/n}} \\ &= \sum_{m=0}^{n-1} e^{2\pi ijm/n} \cdot e^{-2\pi ikm/n} \\ &= \sum_{m=0}^{n-1} e^{2\pi i(j-k)m/n} \end{aligned}$$

Se $j = k$, $p_{jk} = \sum_{m=0}^{n-1} 1 = n$. Senão, p_{jk} forma a série geométrica

$$\begin{aligned} p_{jk} &= \sum_{m=0}^{n-1} (e^{2\pi i(j-k)/n})^m \\ &= \frac{(e^{2\pi i(j-k)/n})^n - 1}{e^{2\pi i(j-k)/n} - 1} \\ &= 0 \end{aligned}$$

uma vez que $e^{2\pi i} = 1$. Assim, $V^{-1} = \frac{1}{n}\overline{V}$. ■

Sendo assim, o inverso da transformada discreta de Fourier é equivalente à transformada discreta, mas trocando $x_k = e^{2\pi ik/n}$ pelo conjugado complexo $e^{-2\pi ik/n}$ na linha 4 e dividindo o valor por n. O algoritmo IDFT é análogo ao DFT, o que resulta num consumo de tempo $\mathrm{O}(n \lg n)$.

3.4 Implementação eficiente

A implementação do algoritmo DFT-Recursivo possui um grande *overhead*. Além da pilha de execução temos vetores sendo devolvidos, etc. Vamos tentar analisar a árvore de recursão do algoritmo de novo para tentar extrair uma abordagem iterativa.

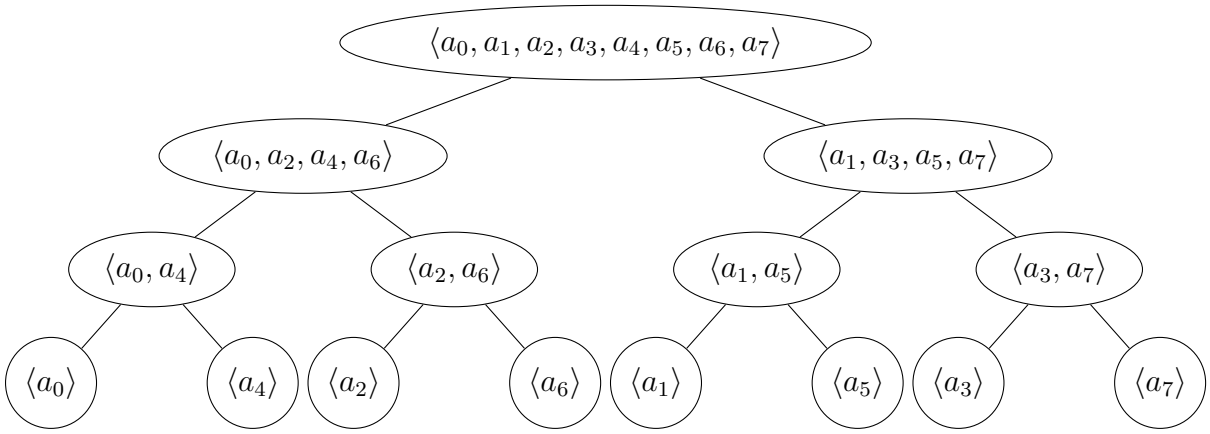

Figura 3.3: *A árvore de chamadas recursivas do algoritmo* DFT-Recursivo *para um polinômio de grau 7.*

Olhando para a árvore de recursão na figura 3.3, percebemos que se processarmos a entrada na ordem que as folhas estão dispostas seria possível fazer uma abordagem *bottom-up* ao contrário da *top-down* proposta na função DFT-Recursivo.

Primeiro computamos o DFT das folhas aos pares num vetor. O vetor agora possui $n/2$ DFTs de dois elementos cada. Na sequência, combinados os $n/2$ DFTs aos pares formando $n/4$ DFTs de quatro elementos. Continuamos até que restem apenas 2 DFTs de $n/2$ elementos para serem combinados no DFT dos n elementos do vetor.

Para implementar esse algoritmo é necessário usar um vetor auxiliar que inicialmente possui os valores do vetor a na ordem que aparecem nas folhas. Essa permutação dos valores de a é conhecida como **permutação reversa de bits**.

Olhemos para a representação binária dos índices da raiz e a das folhas no exemplo da figura 3.3. A sequência dos índices dos coeficientes na raiz usando três bits seria

$$a_{000}, a_{001}, a_{010}, a_{011}, a_{100}, a_{101}, a_{110}, a_{111}$$

Agora reparemos na sequência dos índices dos coeficientes nas folhas

$$a_{000}, a_{100}, a_{010}, a_{110}, a_{001}, a_{101}, a_{011}, a_{111}$$

O coeficiente de índice k vai para a folha com índice igual ao inverso de sua representação binária. Isto é, seja rev(k) o inteiro de $\lg n$ bits que temos ao reverter a representação binária de k, então queremos que a_k esteja na posição A[rev(k)]. Podemos abstrair esse algoritmo para uma função que recebe o vetor a e devolve o vetor A com os coeficientes na ordem das folhas.

PERMUTAÇÃO-REVERSA-DE-BITS(a)
1 $n = a.length$
2 **for** $k = 0$ **to** $n - 1$
3 $\quad A[\text{rev}(k)] = a_k$
4 **return** A

Agora que temos o vetor na ordem necessário vamos analisar como computar o DFT de a iterativamente. Para cada nível s, partindo de 1 (quando estamos nas folhas da árvore combinando DFTs de tamanho 1) até $\lg n$ (quando estamos na raiz combinando DFTs de tamanhos $n/2$) combinamos os DFTs de tamanho 2^{s-1}. Ou seja, o vetor $y^{[0]}$ da função DFT-RECURSIVO que está em $A[k\,..\,k + 2^{s-1} - 1]$ e o vetor $y^{[1]}$ em $A[k + 2^{s-1}\,..\,k + 2^s - 1]$ são combinados em $A[k\,..\,k + 2^s - 1]$.

Note que a função recebe um parâmetro r adicional. Se r for igual a 1, a função FFT-ITERATIVO calcula o DFT de a. Se r for -1, calcula o inverso do DFT de a.

A chamada PERMUTAÇÃO-REVERSA-DE-BITS faz n iterações e para cada uma executa rev(k), que por sua vez reverte uma lista de tamanho $\lg n$. Portanto temos que PERMUTAÇÃO-REVERSA-DE-BITS consome tempo $\Theta(n \lg n)$. As linhas *6-13* são executadas $n/m = n/2^s$ vezes para cada s e as linhas *8-13* são executadas $m/2 = 2^{s-1}$ vezes. Assim,

$$\begin{aligned} T(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \Theta(n \lg n) \end{aligned}$$

Logo, a função FFT-ITERATIVO consome tempo $\Theta(n \lg n)$.

```
FFT-Iterativo(a, r)
 1  A = Permutação-Reversa-de-Bits(a)
 2  n = a.length
 3  for s = 1 to lg n
 4       m = 2^s
 5       ω_m = e^{2rπi/m}
 6       for k = 0 to n − 1 by m
 7            ω = 1
 8            for j = 0 to m/2 − 1
 9                 t = ω × A[k + j + m/2]
10                 u = A[k + j]
11                 A[k + j] = u + t
12                 A[k + j + m/2] = u − t
13                 ω = ω × ω_m
14  if r == -1
15       for k = 0 to n − 1
16            A[k] = A[k]/n
17  return A
```

3.5 Transformada discreta com aritmética modular

Embora a transformada discreta usando raízes complexas da unidade seja eficiente para calcular o DFT de um vetor e, consequentemente, o produto de dois polinômios, os cálculos feitos com números de ponto flutuante podem apresentar erros de arredondamento. Se os coeficientes são valores reais não há o que fazer. Entretanto, para problemas que sabidamente a resposta contém apenas números inteiros positivos seria interessante continuar fazendo as operações apenas com inteiros. Nesta seção vamos estudar uma variante do algoritmo usando aritmética modular chamada NTT (*Number Theoretic Transform*).

O porquê de usarmos as raízes da unidade é óbvio neste ponto do texto. Vamos olhar com mais cuidado quais propriedades das raízes da unidade são de fato necessárias para que o algoritmo para calcular o DFT de um vetor descrito anteriormente funcione corretamente.

1. $z_n^n = 1$
2. $z_n^k \neq 1$ para $1 \leq k < n$
3. Existe uma raiz principal z_n tal que todas as outras raízes da unidade são potências de z_n. Deste modo, as raízes da unidade formam um grupo multiplicativo.

Note que nenhum dos itens acima cita características exclusivas dos números complexos, apenas de grupos. Podemos conseguir esse comportamento fazendo os cálculos para alguns grupos $\mathbb{Z}_p$. Isto é, realizando as operações módulo p.

$$\begin{aligned} z_n^n &= 1 \pmod{p}, \\ z_n^k &\neq 1 \pmod{p} \text{ para } 1 \leq k < n \end{aligned}$$

Teorema 4. *Seja t o menor inteiro positivo tal que $p = 2^t n + 1$ é primo, g um gerador de $\mathbb{Z}_p^*$ e $z_n = g^{2^t} \pmod{p}$. Então, z_n é uma enésima raiz da unidade em $\mathbb{Z}_p^*$.*

Demonstração. Note que

$$\begin{aligned} z^n &= (g^{2^t})^n \pmod p \\ &= g^{2^t n} \pmod p \\ &= g^{p-1} \pmod p \end{aligned}$$

Pelo pequeno teorema de Fermat [11] sabemos que, se p é primo, $a^{p-1} = 1 \pmod p$. Sendo assim, $z_n^n = 1 \pmod p$. Como g é gerador de $\mathbb{Z}_p^*$, $z_n^m \neq 1$ para $1 \leq m < n$. ■

Um resultado imediato também a partir do teorema de Fermat é a existência do inverso z_n.

$$z_n^{p-1} = 1 \pmod p \implies z_n z_n^{p-2} = 1 \pmod p$$

Logo, z_n^{p-2} está em $\mathbb{Z}_p^*$ e é o inverso de z_n.

Também temos o equivalente ao lema de Halving aqui

Teorema 5. *Se n é um inteiro positivo par e z_n é uma enésima raiz da unidade de $\mathbb{Z}_p^*$, então os quadrados das enésimas raízes da unidade de $\mathbb{Z}_p^*$ são as $n/2$-ésimas raízes da unidade de $\mathbb{Z}_p^*$.*

Demonstração.

$$\begin{aligned} (z_n^2)^m = z_n^n = 1 \pmod p, &\quad \text{para } m = n/2 \\ (z_n^2)^t = z_n^{2t} \neq 1 \pmod p, &\quad 1 \leq t < m. \end{aligned}$$

■

Dos teoremas anteriores temos que todas as 2^t raízes da unidade existem em $\mathbb{Z}_p^*$ bem como seus quadrados também o são. Sendo assim, todas as propriedades para o algoritmo estão mantidas se tomarmos as raízes da unidade em $\mathbb{Z}_p^*$ para $p = 2^t n + 1$ primo. Basta alterar o código para que os cálculos sejam feitos usando aritmética modular.

O leitor deve se perguntar como calcular os valores de t e g. É esperado que testar candidatos para t seja $\mathrm{O}(\lg n)$, mas achar um gerador não é uma tarefa simples e ambos os testes estão fora do escopo deste texto. Para programação competitiva, os problemas envolvendo FFT geralmente possuem entradas de ordem 10^5. Portanto, basta guardar as constantes $t = 20, p = 7340033 = 7 \cdot 2^{20} + 1$ grande o suficiente e $g = 5$.

É importante salientar que agora a função recebe um booleano *reverse* que indica se deve ser calculado a ida ou a volta da transformada.

Como a única diferença entre o FFT e o NTT são os valores sendo multiplicados, o consumo de tempo da função NTT-Iterativo é $\mathrm{O}(n \lg n)$.

NTT-ITERATIVO$(a, reverse)$

```
 1  A = PERMUTAÇÃO-REVERSA-DE-BITS(a)
 2  n = a.length
 3  for s = 1 to lg n
 4      m = 2^s (mod p)
 5      if reverse == TRUE
 6          z_m = g (mod p)
 7      else
 8          z_m = g^{p-2} (mod p)
 9      for j = s to t - 1
10          z_m = z_m × z_m (mod p)
11      for k = 0 to n - 1 by m
12          z = 1
13          for j = 0 to m/2 - 1
14              t = z × A[k + j + m/2] (mod p)
15              u = A[k + j]
16              A[k + j] = u + t (mod p)
17              A[k + j + m/2] = u - t (mod p)
18              z = z × z_m (mod p)
19  if reverse == TRUE
20      n^{-1} = n^{p-2}
21      for k = 0 to n - 1
22          A[k] = A[k] · n^{-1} (mod p)
23  return A
```

Capítulo 4

Problemas

Em programação competitiva problemas computacionais são apresentados a equipes de programadores. Há um juiz eletrônico que é um programa que avalia os programas submetidos pelas equipes. O juiz submete uma bateria de testes aos programas das equipes. Frequentemente uma das dificuldades das equipes em terem seu programas considerados corretos pelo juiz é o limite de tempo dado aos programas para resolverem os testes.

Aqui, eficiência toma o centro da discussão. Veremos neste capítulo alguns problemas que usam a Transformada Rápida de Fourier na solução. Cada uma das seções apresentam uma problema e a discussão da resposta.

4.1 Multiplicação rápida de polinômios

Descrição

Sejam $A(x)$ e $B(x)$ dois polinômios de grau $n-1$ e $m-1$ respectivamente. Suponha que m seja menor que n, sem perda de generalidade. Queremos um polinômio $C(x)$ que seja igual ao produto $A(x) \cdot B(x)$ para todo x.

Discussão

O DFT de A é o valor de A calculado nas n raízes da unidade, portanto, pelo teorema da convolução [9] se tivermos o DFT de A e B podemos computar o DFT de C apenas multiplicando cada coordenada dos vetores. Entretanto, o grau do polinômio C é $n+m-2$: e pelo teorema fundamental da álgebra, para definir um polinômio de grau $k-1$ são necessários k pontos. Uma maneira de conseguir o DFT de A e B em mais pontos é aumentar o grau de ambos os polinômios, basta que o vetor de coeficientes de ambos seja completado com zeros nas potências maiores.

O algoritmo para multiplicar dois polinômios consiste em completar o de menor grau até que os dois fiquem do mesmo tamanho, completar ambos os polinômios para a próxima potência de dois maior ou igual a n, computar o DFT de ambos os vetores de coeficientes, multiplicar os pontos de mesma coordenada e então calcular o IDFT do vetor resultante.

Solução

A função Multiplica-Polinômios recebe dois vetores de coeficientes a e b e devolve os coeficientes do produto dos polinômios a por b.

O primeiro passo é garantir que os vetores tenham o mesmo tamanho e que o tamanho seja uma potência de dois. A função pushback(a, x) recebe um vetor a e um valor x e insere x no final de a.

MULTIPLICA-POLINÔMIOS(a, b)

```
1  n = a.length // a.length ≥ b.length
2  while b.length < n
3       pushback(b, 0) // Adiciona um zero no fim do vetor b
4  while n&(n − 1) ≠ 0 // Enquanto n não é uma potência de 2
5       n = n + 1
6       pushback(a, 0)
7       pushback(b, 0)
8  dft-a = DFT(a)
9  dft-b = DFT(b)
10 for k = 0 to n − 1
11      dft-c_k = dft-a_k · dft-b_k
12 c = IDFT(dft-c)
13 return c
```

Como o laço da linha 5 no máximo dobra o tamanho dos vetores e cada operação das linhas *6-8* é $\Theta(1)$ o algoritmo consome tempo $\Theta(n \lg n)$.

4.2 Todas as possíveis somas

Descrição

São dadas duas listas a e b com s e t inteiros, respectivamente. Queremos saber todas as somas possíveis entre um elemento de a e um elemento de b e de quantas formas esse valor pode ser obtido. Por exemplo, para $a = (1, 2, 3)$ e $b = (2, 4)$ temos que 3 pode ser obtido de uma maneira, 4 de uma maneira, 5 de duas maneiras, 6 de uma maneira e 7 de uma maneira.

$$3 = 1 + 2$$
$$5 = 1 + 4$$
$$4 = 2 + 2$$
$$6 = 2 + 4$$
$$5 = 3 + 2$$
$$7 = 3 + 4$$

Discussão

Calcular todas as somas e quantidades uma a uma é trivial. Como fazer de maneira eficiente? Note que se k aparece p vezes em A e j aparece q vezes em B, podemos obter $k+j$ de pq formas diferentes. Se j e k fossem expoentes de uma variável x e p e q seus respectivos coeficientes, ao calcularmos o produto $px^k \cdot qx^j$ temos pqx^{k+j}. Ou seja, a soma aparece no expoente e a quantidade de vezes que essa soma pode ser obtida aparece num coeficiente.

Sendo assim, vamos definir um polinômio $A(x)$ de modo que, se k aparece p vezes em a, px^k é um monômio de $A(x)$. Isto é, se $a = (1, 2, 2)$, $A(x) = x^1 + 2x^2$. Construímos também um polinômio $B(x)$ para a lista b análogo ao descrito para $A(x)$. O produto dos polinômios $A(x)$ e $B(x)$ nos dá um polinômio $C(x)$ que o coeficiente k diz quantas duplas somam k nas listas a e b.

O algoritmo consome tempo $\mathrm{O}(n \lg n)$, onde n é o maior inteiro das listas a e b.

4.3 Distância de Hamming para todas as substrings

Descrição

A **distância de Hamming** entre duas strings a e b de mesmo tamanho é definida como o número de posições i na qual a_i é diferente de b_i. Por exemplo, a distância de Hamming entre "01011" e "00010" é 1.

Uma string é dita ***k*-vizinha** de uma outra string se e somente se elas são do mesmo tamanho e a distância de Hamming entre as duas é menor ou igual a k. Dadas duas strings binárias a e b queremos saber quantas substrings de a são k-vizinhas de b.

Discussão

Para facilitar a representação digamos que a tenha tamanho n, b tenha tamanho m e $m \leq n$. Estamos interessados na distância de Hamming d_i para $i = 0, 1, \ldots, n - m - 1$

$$d_i = \sum_{j=0}^{m-1} |a_{i+j} - b_j|$$

Vale notar que como o alfabeto é binário, portanto

$$\begin{aligned} d_i &= \sum_{j=0}^{m-1} |a_{i+j} - b_j| \\ &= \sum_{j=0}^{m-1} (a_{i+j} - b_j)^2 \\ &= \sum_{j=0}^{m-1} a_{i+j}^2 + b_j^2 - 2a_{i+j}b_j \\ &= \sum_{j=0}^{m-1} a_{i+j}^2 + \sum_{j=0}^{m-1} b_j^2 - 2\sum_{j=0}^{m-1} a_{i+j}b_j \end{aligned}$$

O termo $\sum_{j=0}^{m-1} b_j^2$ é constante e $\sum_{j=0}^{m-1} a_{i+j}^2$ pode ser calculado em tempo constante usando somas de prefixos. A soma $\sum_{j=0}^{m-1} a_{i+j}b_j$ é muito parecida com a convolução dos vetores a e b. O que muda é o fato da soma dos índices não ser constante e sim a diferença. Vamos transformar os vetores para que a soma seja constante e essa soma possa ser calculada usando a multiplicação rápida de polinômios vista anteriormente.

Seja o vetor c o reverso de b. Ou seja $c_i = b_{m-i-1}$. Assim,

$$\widetilde{d}_i = \sum_{j=0}^{m-1} a_{i+j}b_j = \sum_{j=0}^{m-1} a_{i+j}c_{m-j-1}$$

Dessa forma a soma dos índices é constante, isto é, é uma convolução. Ao calcular o produto dos polinômios $D(x) = A(x) \cdot C(x)$ o valor de $\widetilde{d}_i$ será c_{m+i-1}.

4.4 Número de somas distintas

Descrição

Este problema apareceu na fase regional brasileira da Maratona de Programação em 2017 e pode ser consultado em https://www.urionlinejudge.com.br/judge/pt/problems/view/2669.

Uma cadeia ponderada é definida sobre um alfabeto Σ e uma função f que atribui um peso a cada caractere do alfabeto. Assim, podemos definir o peso de uma cadeia s como a soma dos pesos de todos os caracteres em s.

Vários problemas da bioinformática podem ser formalizados como problemas em cadeias ponderadas. Um exemplo é a espectrometria de massa de proteínas, uma técnica que permite identificar proteínas de forma bastante eficiente. Podemos representar cada aminoácido por um caractere distinto e uma proteína é representada pela cadeia de caracteres relativos aos amino ácidos que a compõe.

Uma das aplicações da espectrometria de massa de proteínas são buscas em bancos de dados. Para isso a cadeia que representa a proteína é dividida em sub-cadeias, a massa de cada sub-cadeia é determinada, e a lista de massas é comparada com um banco de dados de proteínas. Um dos desafios para essa técnica é lidar com cadeias muito grandes de caracteres, que podem ter várias possíveis sub-cadeias. A quantidade de sub-cadeias selecionadas é fundamental para obter bons resultados.

Considerando que s é formada por letras minúsculas e cada letra tem um peso diferente entre 1 e 26: a letra a tem peso 1, a letra b tem peso 2 e assim por diante, queremos determinar a quantidade de pesos distintos encontrada ao avaliar os pesos de todas as sub-cadeias não vazias de caracteres consecutivos de s.

Por exemplo *abbab* tem 8 pesos distintos

$$\begin{aligned} a &= 1 \\ b &= 2 \\ a+b = b+a &= 3 \\ b+b &= 4 \\ a+b+b = b+b+a = b+a+b &= 5 \\ a+b+b+a &= 6 \\ b+b+a+b &= 7 \\ a+b+b+a+b &= 8 \end{aligned}$$

Discussão

Para resolver este problema não vamos trabalhar diretamente com a cadeia s mas sim com um outro vetor *sum* que armazena a soma de prefixos de s. Ou seja, sum_i é a soma dos pesos de $s_0, s_1, \ldots, s_{i-1}$. Um valor v é contado na cadeia s se existem índices $r > l$ tal que $sum_r - sum_l = v$. Para a cadeia *abbab* o vetor *sum* é $[0, 1, 3, 5, 6, 8]$.

Vamos definir um polinômio $A(x)$ a partir dos prefixos de s de modo que

$$a_k = \begin{cases} 1, & \text{se } \exists i \text{ tal que } sum_i = k \\ 0, & \text{caso contrário} \end{cases}$$

Também definimos um polinômio $B(x)$ usando *sum*. Entretanto, diferente de $A(x)$ que usa

os prefixos de s, $B(x)$ usará os sufixos $sum_n - sum_i$, onde n é o tamanho da string s.

$$b_k = \begin{cases} 1, & \text{se } \exists i \text{ tal que } sum_n - sum_i = k \\ 0, & \text{caso contrário} \end{cases}$$

Deste modo, para $s = abbab$,

$$A(x) = x^0 + x^1 + x^3 + x^5 + x^6 + x^8 \text{ e}$$
$$B(x) = x^0 + x^2 + x^3 + x^5 + x^7 + x^8$$

Tomemos agora o polinômio $C(x) = A(x) \cdot B(x)$. Lembrando a equação (2.6), c_k é $a_i b_j$ para todo i, j tal que $i + j = k$. Reescrevendo os índices i e j temos

$$\begin{aligned} k &= i + j \\ &= sum_p + sum_n - sum_q \end{aligned}$$

Repare que sum_n é uma constante, então

$$\begin{aligned} k &= i + j \\ &= constante + sum_p - sum_q \end{aligned}$$

Logo, para todo coeficiente $c_k > 0$ com valor v significa que podemos obter $v - sum_n$ usando a diferença de duas somas de prefixos. Se contarmos quantos coeficientes c_k são maiores que zero para $k > sum_n$ saberemos quantas cadeias ponderadas distintas podemos obter em s.

Capítulo 5

Conclusões

Problemas que usam a transformada rápida de Fourier (FFT) aparecem cada vez mais nas competições de programação competitiva. Nossa intenção com este texto foi apresentar a FFT e suas aplicações aos participantes de programação competitiva. Procuramos descrever os algoritmos e as principais ideias para o entendimento dessa técnica. Esperamos ter apresentado o tema com clareza e destacado sua importância para que mais leitores possam se interessar tanto pela programação competitiva quanto pela Transformada de Fourier e aplicar o conteúdo aqui apresentado. Embora o texto exista para servir de material para competidores da Maratona de Programação, esperamos que também seja útil para leitores que estão apenas interessados em algoritmos em geral e que auxilie na sua formação. Para os participantes de programação competitiva deixamos no apêndice um trecho de código que pode ser anexado ao "caderno" de códigos de time.

FFT em particular e análise de Fourier em geral são tópicos muito abrangentes. Não poderíamos deixar de mencionar o projeto Fast Fourier Transform in the West (FFTW) [6]. FFTW é uma biblioteca em C para calcular a transformada discreta de Fourier. O código desse projeto utiliza técnicas algorítmicas engenhosas e de *hacking*. Apenas as referências contidas nessa página já mostram o quão vasto e fascinante é esse tópico.

Apêndice A

Implementações

É comum que times e até mesmo competidores individuais tenham um caderno com códigos prontos de diversos algoritmos para usar durante competições se necessário. Para aqueles que estão interessados em usar a transformada de Fourier em seus cadernos vamos deixar aqui a nossa implementação dos algoritmos FFT e NTT ambas em C++.

A função BitReversalPermutation(a) usa os templates do C++ funcionando para ambos os algoritmos.

```
1  template <typename T>
2  vector<T> BitReversalPermutation(const vector<T>& a) {
3    int n = a.size();
4    int logn = __builtin_ctz(n);
5    vector<T> v(n);
6    for (int k = 0; k < n; k++) {
7      int rev = 0; // rev = reverse(k)
8      for (int j = 0; j < logn; j++)
9      if (k&(1<<j)) rev |= (1<<(logn - 1 - j));
10     v[rev] = a[k];
11   }
12   return v;
13 }
```

O FFT usará *long double* e o tipo nativo *complex* da STL. A função recebe o vetor a e um inteiro r e devolve DFT(a) caso r seja igual a 1. Devolve IDFT(a) se r for -1.

```
1  typedef complex<long double> Complex;
2  const long double PI = acos(-1.0L);
3
4  vector<Complex> FFT(vector<Complex> a, const int r) {
5    int n = a.size();
6    int logn = __builtin_ctz(n); // n is a power of two
7    vector<Complex> v = BitReversalPermutation<Complex>(a);
8    for (int s = 1; s <= logn; s++) {
9      int m = (1 << s);
10     Complex wm(cos(2.0 * r * PI / m), sin(2.0 * r * PI / m));
11     for (int k = 0; k < n; k += m) {
12       Complex w = 1;
13       for (int j = 0; 2*j < m; j++) {
14         Complex t = w * v[k + j + m/2];
15         Complex u = v[k + j];
16         v[k + j] = u + t;
17         v[k + j + m/2] = u - t;
18         w *= wm;
19       }
20     }
```

```
21      }
22      if (r == -1)
23      for (Complex& c : v) c /= n;
24      return v;
25    }
```

Já o NTT será apenas com inteiros, como esperado, e usará um módulo p suficientemente grande para entradas na ordem de 10^5. Outra diferença é que a função recebe um booleano *reverse* e devolve o DFT de a se *reverse* for FALSO. Do contrário, devolve o IDFT de a.

A função FEXP calcula n^{p-2} usando exponenciação rápida módulo p.

```
1   const int mod = 7340033;
2   const int g = 5; // g^{2^{20}} = 1 mod p
3   const int gr = 4404020; // g^{p-2} mod p
4   const int t = 20;
5
6   vector<int> NTT(vector<int> a, bool reverse) {
7     int n = a.size();
8     int logn = __builtin_ctz(n);
9     vector<int> v = BitReversalPermutation<int>(a);
10    for (int s = 1; s <= logn; s++) {
11      int m = 1 << s;
12      int zm = reverse ? gr : g;
13      int cont = 0;
14      for (int i = s; i < t; i++)
15      zm = (int)(1LL * zm * zm % mod);
16      for (int k = 0; k < n; k += m) {
17        int z = 1;
18        for (int j = 0; 2 * j < m; j++) {
19          int t = (int)(1LL * v[k + j + m/2] * z % mod);
20          int u = v[k + j];
21          v[k + j] = u + t < mod ? u + t : u + t - mod;
22          v[k + j + m/2] = u - t >= 0 ? u - t : u - t + mod;
23          z = (int)(1LL * z * zm % mod);
24        }
25      }
26    }
27    if (reverse == true) {
28      int n_1 = fexp(n, mod-2);
29      for (int& x : v) x = (int) (1LL * x * n_1 % mod) ;
30    }
31    return v;
32  }
```

Bibliografia

[1] **Cooley e Tukey(1965)** James W. Cooley e John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, páginas 297–301. Citado na pág. 1

[2] **Cooley** *et al.***(1967)** James W. Cooley, Peter A.W. Lewis e Peter D. Welch. Historical notes on fast Fourier transform. *Proceedings of IEEE*, 55(10):1675–1677. Citado na pág. 1

[3] **Cooley** *et al.***(1969)** James W. Cooley, Peter A.W. Lewis e Peter D. Welch. The fast Fourier transform and its applications. *IEEE*, páginas 27–34. Citado na pág. 14

[4] **Cormen** *et al.***(2009)** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edição. ISBN 0262033844, 9780262033848. Citado na pág. 7

[5] **Fourier(1822)** Jean-Baptiste Joseph Fourier. *Théorie analytique de la chaleur*. Firmin Didot Père et Fils. Citado na pág. 1

[6] **Frigo e Johnson()** Matteo Frigo e Steven G. Johnson. The fastest Fourier transform in the west. http://www.fftw.org/. acessado em 22/11/2018. Citado na pág. 27

[7] **Heideman** *et al.***(1985)** M.T. Heideman, D.H. Johnson e C.S. Burrus. Gauss and the history of the fast Fourier transform. *Archive for History of Exact Sciences*, páginas 265–277. Citado na pág. 1

[8] **Wikipedia(2018)** Wikipedia. Algebraic variety. https://en.wikipedia.org/wiki/Algebraic_variety, novembro 2018. acessado em 20/11/2018. Citado na pág. 3

[9] **Wikipedia(2018)** Wikipedia. Convolution theorem. https://en.wikipedia.org/wiki/Convolution_theorem, novembro 2018. acessado em 22/11/2018. Citado na pág. 21

[10] **Wikipedia(2018)** Wikipedia. List of the 72 names on the Eiffel tower. https://en.wikipedia.org/wiki/List_of_the_72_names_on_the_Eiffel_Tower, novembro 2018. acessado em 22/11/2018. Citado na pág. 1

[11] **Wikipedia(2018)** Wikipedia. Fermat's little theorem. https://en.wikipedia.org/wiki/Fermat's_little_theorem, novembro 2018. acessado em 20/11/2018. Citado na pág. 18

[12] **Wikipedia(2018)** Wikipedia. Lagrange polynomial. https://en.wikipedia.org/wiki/Algebraic_variety, novembro 2018. acessado em 20/11/2018. Citado na pág. 6

[13] **Wikipedia(2018)** Wikipedia. Taylor series. https://en.wikipedia.org/wiki/Lagrange_polynomial, novembro 2018. acessado em 20/11/2018. Citado na pág. 3