Árvore de Fenwick

Aplicações e variantes

Prof. Edson Alves – UnB/FGA

Sumário

Aplicações da árvore de Fenwick

Condições necessárias para o uso de uma árvore de Fenwick

- Para utilizar uma árvore de Fenwick para realizar a operação $\odot$ em um intervalo de índices $[i, j]$ da sequência a_k, é necessário que esta operação tem duas propriedades
- A primeira propriedade é a associatividade: para quaisquer $x, y, z \in a_k$, deve valer que

$$(x \odot y) \odot z = x \odot (y \odot z)$$

- A segunda propriedade é a invertibilidade: para qualquer $x \in a_k$, deve existir um valor y tal que $x \odot y = I$, onde I é o elemento neutro/identidade da operação $\odot$
- Como exemplos de operações que tem ambas propriedades temos a adição e a multiplicação de racionais, a adição de matrizes e o ou exclusivo (*xor*)

Range product query

- É possível utilizar uma árvore de Fenwick para realizar o *range product query*, de forma semelhante ao que foi feito na adição
- Neste caso, a operação de atualização corresponde a multiplicação de um elemento a_i pela constante k
- Dois pontos devem ser observados: em primeiro lugar, é preciso tratar com cuidado a possibilidade de *overflow*
- Em segundo lugar, o zero não é invertível em relação à multiplicação: logo a quantidade de zeros deve ser mantida à parte, em outra árvore de Fenwick, e as ambas árvores devem ser combinadas para produzir o resultado correto

Implementação de uma árvore de Fenwick para produtos

```
template<typename T>
class BITree
{
private:
    T N;
    vector<T> ft, zs;                                    // zi = 1 se ai = 0

    T LSB(const T& n) { return n & -n; }

public:
    BITree(int n) : N(n), ft(N + 1, 1), zs(N + 1, 0) { }

    T RPQ(int i, int j)
    {
        auto p = RPQ(j) / RPQ(i - 1);
        auto z = RSQ(i, j);

        return z ? 0 : p;
    }
```

Implementação de uma árvore de Fenwick para produtos

```cpp
    void update(int i, const T& v)
    {
        if (v)
            multiply(i, v);
        else if (RSQ(i, i) == 0)
            add(i, 1);
    }

private:
    int RPQ(int i)
    {
        int prod = 1;

        while (i) {
            prod *= ft[i];
            i -= LSB(i);
        }

        return prod;
    }
```

Implementação de uma árvore de Fenwick para produtos

```
    int RSQ(int i, int j)
    {
        return RSQ(j) - RSQ(i - 1);
    }

    int RSQ(int i)
    {
        int sum = 0;

        while (i)
        {
            sum += zs[i];
            i -= LSB(i);
        }

        return sum;
    }
```

Implementação de uma árvore de Fenwick para produtos

```
    void multiply(int i, int v)
    {
        while (i <= N)
        {
            ft[i] *= v;
            i += LSB(i);
        }
    }

    void add(int i, int v)
    {
        while (i <= N)
        {
            zs[i] += v;
            i += LSB(i);
        }
    }
};
```

Contagem de inversões

- Árvores de Fenwick também podem ser utilizadas para contar o número de inversões em uma sequência de inteiros positivos a_k
- Uma inversão acontece quando $a_i > a_j$ e $i < j$
- Seja M é o valor máximo que um elemento a_i pode assumir
- A cada elemento a_i da sequência, com $i = 1, 2, \ldots, N$, $RSQ(a_i + 1, M)$ conterá o número de elementos já inseridos na árvore que são estritamente maiores do que a_i, isto é, o número de inversões do tipo (i, j), com $j < i$
- Após esta contagem, o valor a_i deve ser incrementado em uma unidade na árvore
- Esta abordagem tem complexidade $O(N \log M)$

Implementação da contagem de inversões

```cpp
template<typename T>
long long count_inversions(const vector<T>& as)
{
    T M = *max_element(as.begin(), as.end());
    BITree<T> ft(M);

    long long inversions = 0;

    for (size_t i = 0; i < as.size(); ++i)
    {
        inversions += ft.RSQ(as[i] + 1, M);
        ft.add(as[i], 1);
    }

    return inversions;
}
```

Compressão do domínio

- Se, no problema anterior, M for grande (por exemplo, $M \geq 10^9$), não é possível contruir uma árvore de Fenwick que comporte este número de elementos
- Contudo, se N for pequeno (por exemplo, $N \leq 10^5$), apenas N dentre todos os M valores aparecerão na árvore
- Assim, pode se construir um mapeamento $f : T \to \mathbb{N}$ entre estes N valores e os N primeiros números naturais de modo que $f(a) < f(b)$ se $a < b$
- Daí a árvore de Fenwick armazenaria e manipularia os representantes, e não os reais valores da sequência, contando ainda assim as inversões de forma correta

Exemplo de compressão de domínio

```
template<typename T>
long long count_inversions_compression(const vector<T>& as)
{
    set<T> xs(as.begin(), as.end());
    map<T, int> f;
    size_t N = 0;

    for (const auto& x : xs)
        f[x] = ++N;

    BITree<T> ft(N);
    long long inversions = 0;

    for (const auto& a : as) {
        inversions += ft.RSQ(f[a] + 1, N);
        ft.add(f[a], 1);
    }

    return inversions;
}
```

Variações da árvore de Fenwick

Range update, point query

- Em sua proposta original a árvore de Fenwick responde *queries* no intervalo $[i, j]$ e atualiza os valores em pontos i
- É possível somar um valor fixo k em todos os elementos a_k cujos índices estão no intervalo $[i, j]$ (*range update*) com complexidade $O(\log N)$
- Para isso é necessário reinterpretar os nós da árvore de Fenwick de tal forma que o elemento t_k armazenará um valor x que deve ser adicionado a todos os elementos a_i tais que $i \geq k$
- Assim, uma atualização `update(i, j, k)` será feita por meio de duas soma pontuais: `add(i, k)` e `add(j + 1, -k)`
- O efeito desta atualização é apresentado abaixo:

$$\ldots, a_{i-1}, a_i + k, a_{i+1} + k, \ldots, a_{j-1} + k, a_j + k, a_{j+1}, \ldots$$

- Para determinar o valor do elemento a_i, é preciso fazer uma RSQ deste o início até o ponto i
- Em notação matemática,

$$a_i = RSQ(1, i) = \sum_{k=1}^{i} t_k$$

- Deste modo, a *query* retorna o valor de um único ponto (*point query*)
- É possível combinar *range updates* com *range queries*, usando-se duas árvores de Fenwick

Implementação de *range update* com *point query*

```
template<typename T>
class BITree {
public:
    // É preciso adicionar um elemento sentinela na última posição para evitar o
    // erro 'off-by-one' na implementação de range_add()
    BITree(size_t n) : ts(n + 2, 0), N(n) {}

    ll value_at(int i) { return RSQ(i); }

    void range_add(size_t i, size_t j, ll x)
    {
        add(i, x);
        add(j + 1, -x);
    }

private:
    vector<T> ts;
    size_t N;

    int LSB(int n) { return n & (-n); }
```

Implementação de *range update* com *point query*

```
    ll RSQ(int i)
    {
        ll sum = 0;

        while (i >= 1) {
            sum += ts[i];
            i -= LSB(i);
        }

        return sum;
    }

    void add(size_t i, ll x)
    {
        while (i <= N) {
            ts[i] += x;
            i += LSB(i);
        }
    }
};
```

Fenwick Tree bidimensional

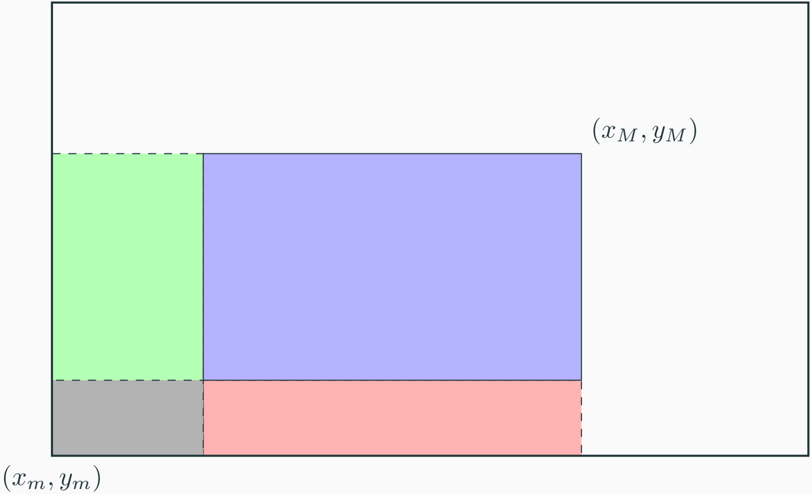
- É possível estender o conceito de árvore de Fenwick para duas dimensões
- Cada nó $t_{r,s}$ da árvore de Fenwick bidimensional armazenará a soma de todos os elementos $a_{i,j}$ da matriz A cujos índices pertençam aos intervalos $I_r = (r - p(r) + 1, r)$ e $I_s = (s - p(s) + 1, s)$, respectivamente
- Esta interpretação permite uma implementação eficiente tanto da rotina de atualização quando da rotina de soma de intervalo
- A atualização consiste em somar um valor fixo v no elemento $a_{i,j}$ da matriz
- Devem ser atualizados, portanto, todos os intervalos que contenham tal elemento, o que pode ser feito com um laço duplo

Fenwick Tree bidimensional

- $RSQ(i, j)$ é a soma todos os elementos no retângulo definido pelos vértices opostos $(1, 1)$ e (i, j) e é a extensão natural da rotina $RSQ(i)$ da árvore unidimensional
- Para se determinar $RSQ(x_m, y_m, x_M, y_M)$, deve-se utilizar o Princípio da Inclusão-Exclusão, isto é,

$$\begin{aligned} RSQ(x_m, y_m, x_M, y_M) &= RSQ(x_M, y_M) - RSQ(x_m - 1, y_M) \\ &\quad - RSQ(x_M, y_m - 1) + RSQ(x_m - 1, y_m - 1) \end{aligned}$$

Visualização de $RSQ(x_m, y_m, x_M, y_M)$



Implementação da árvore de Fenwick bidimensional

```
template<typename T>
class BITree2D {
public:
    vector<vector<T>> ft;
    int N;

    BITree2D(size_t n) : ft(n + 1, vector<T>(n + 1, 0)), N(n) {}

    // Range query
    T RSQ(int a, int b, int c, int d) {
        return RSQ(c, d) - RSQ(c, b-1) - RSQ(a-1, d) + RSQ(a-1, b-1);
    }

    // Point update
    void add(int x, int y, T v)
    {
        for (int i = x; i <= N; i += LSB(i))
            for (int j = y; j <= N; j += LSB(j))
                ft[i][j] += v;
    }
```

Implementação da árvore de Fenwick bidimensional

```
private:
    int LSB(int n) { return n & -n; }

    T RSQ(int a, int b)
    {
        int sum = 0;

        for (int i = a; i > 0; i -= LSB(i))
            for (int j = b; j > 0; j -= LSB(j))
                sum += ft[i][j];

        return sum;
    }
};
```

Referências

1. **FENWICK**, Peter M. A New Data Structure for Comulative Frequency Tables, Journal of Software: Pratice and Experience, volume 24, issue 3, 1994.
2. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2010.
3. <COD>CAD. BIT 2D, acesso em 13/05/2019.
4. GeeksForGeeks. Binary Indexed Tree: Range Updates and Point Queries, acesso em 13/05/2019.
5. Wikipedia. Fenwick Tree, acesso em 06/05/2019.