

Árvore de Fenwick

Aplicações e variantes

Prof. Edson Alves – UnB/FGA

1. Aplicações da árvore de Fenwick
2. Variações da árvore de Fenwick

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*)

- 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

```
1 template<typename T>
2 class BITree
3 {
4 private:
5     T N;
6     vector<T> ft, zs;           // zi = 1 se ai = 0
7
8     T LSB(const T& n) { return n & -n; }
9
10 public:
11     BITree(int n) : N(n), ft(N + 1, 1), zs(N + 1, 0) { }
12
13     T RPQ(int i, int j)
14     {
15         auto p = RPQ(j) / RPQ(i - 1);
16         auto z = RSQ(i, j);
17
18         return z ? 0 : p;
19     }
```

Implementação de uma árvore de Fenwick para produtos

```
21 void update(int i, const T& v)
22 {
23     if (v)
24         multiply(i, v);
25     else if (RSQ(i, i) == 0)
26         add(i, 1);
27 }
28
29 private:
30 int RPQ(int i)
31 {
32     int prod = 1;
33
34     while (i) {
35         prod *= ft[i];
36         i -= LSB(i);
37     }
38
39     return prod;
40 }
```

Implementação de uma árvore de Fenwick para produtos

```
42  int RSQ(int i, int j)
43  {
44      return RSQ(j) - RSQ(i - 1);
45  }
46
47  int RSQ(int i)
48  {
49      int sum = 0;
50
51      while (i)
52      {
53          sum += zs[i];
54          i -= LSB(i);
55      }
56
57      return sum;
58  }
```


Implementação de uma árvore de Fenwick para produtos

```
60 void multiply(int i, int v)
61 {
62     while (i <= N)
63     {
64         ft[i] *= v;
65         i += LSB(i);
66     }
67 }
68
69 void add(int i, int v)
70 {
71     while (i <= N)
72     {
73         zs[i] += v;
74         i += LSB(i);
75     }
76 }
77 };
```

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 $i = 1, 2, \dots, N$, $RSQ(a_i + 1, M)$ conterà 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

```
1 template<typename T>
2 long long count_inversions(const vector<T>& as)
3 {
4     T M = *max_element(as.begin(), as.end());
5     BITree<T> ft(M);
6
7     long long inversions = 0;
8
9     for (size_t i = 0; i < as.size(); ++i)
10    {
11        inversions += ft.RSQ(as[i] + 1, M);
12        ft.add(as[i], 1);
13    }
14
15    return inversions;
16 }
```

- 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 $f : T \rightarrow \mathbb{N}$ 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

```
1  template<typename T>
2  long long count_inversions_compression(const vector<T>& as)
3  {
4      set<T> xs(as.begin(), as.end());
5      map<T, int> f;
6      size_t N = 0;
7
8      for (const auto& x : xs)
9          f[x] = ++N;
10
11     BITree<T> ft(N);
12     long long inversions = 0;
13
14     for (const auto& a : as) {
15         inversions += ft.RSQ(f[a] + 1, N);
16         ft.add(f[a], 1);
17     }
18
19     return inversions;
20 }
```

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 $\text{update}(i, j, k)$ será feita por meio de duas soma pontuais: $\text{add}(i, k)$ e $\text{add}(j + 1, -k)$
- O efeito desta atualização é apresentado abaixo:

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

- 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*

```
1 template<typename T>
2 class BITree {
3 public:
4     // É preciso adicionar um elemento sentinela na última posição para evitar o
5     // erro 'off-by-one' na implementação de range_add()
6     BITree(size_t n) : ts(n + 2, 0), N(n) {}
7
8     ll value_at(int i) { return RSQ(i); }
9
10    void range_add(size_t i, size_t j, ll x)
11    {
12        add(i, x);
13        add(j + 1, -x);
14    }
15
16 private:
17     vector<T> ts;
18     size_t N;
19
20     int LSB(int n) { return n & (-n); }
```

Implementação de *range update com point query*

```
22  ll RSQ(int i)
23  {
24      ll sum = 0;
25
26      while (i >= 1) {
27          sum += ts[i];
28          i -= LSB(i);
29      }
30
31      return sum;
32  }
33
34  void add(size_t i, ll x)
35  {
36      while (i <= N) {
37          ts[i] += x;
38          i += LSB(i);
39      }
40  }
41  };
```

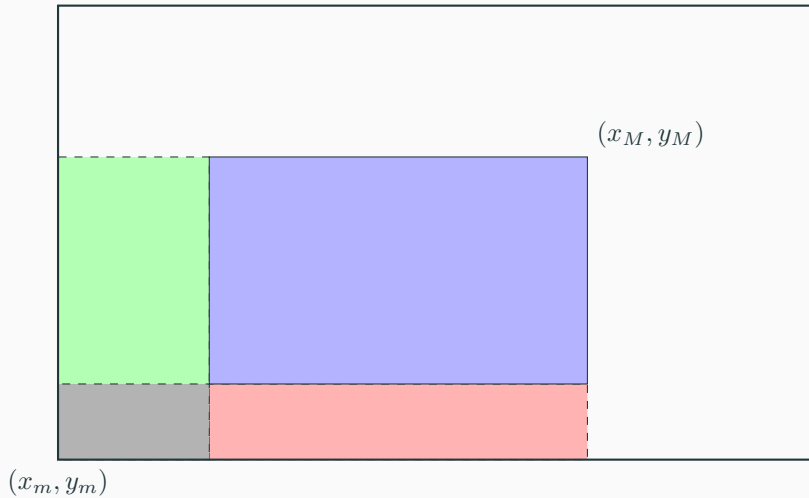
Fenwick Tree bidimensional

- 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

- $RSQ(i, j)$ 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,

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

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



Implementação da árvore de Fenwick bidimensional

```
1 template<typename T>
2 class BITree2D {
3 public:
4     vector<vector<T>> ft;
5     int N;
6
7     BITree2D(size_t n) : ft(n + 1, vector<T>(n + 1, 0)), N(n) {}
8
9     // Range query
10    T RSQ(int a, int b, int c, int d) {
11        return RSQ(c, d) - RSQ(c, b-1) - RSQ(a-1, d) + RSQ(a-1, b-1);
12    }
13
14    // Point update
15    void add(int x, int y, T v)
16    {
17        for (int i = x; i <= N; i += LSB(i))
18            for (int j = y; j <= N; j += LSB(j))
19                ft[i][j] += v;
20    }
```

Implementação da árvore de Fenwick bidimensional

```
22 private:
23     int LSB(int n) { return n & -n; }
24
25     T RSQ(int a, int b)
26     {
27         int sum = 0;
28
29         for (int i = a; i > 0; i -= LSB(i))
30             for (int j = b; j > 0; j -= LSB(j))
31                 sum += ft[i][j];
32
33         return sum;
34     }
35 };
```

1. **FENWICK**, Peter M. [A New Data Structure for Cumulative Frequency Tables](#), Journal of Software: Practice 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.