Árvore de Fenwick

Definição, *RSQ* e *update*

Prof. Edson Alves – UnB/FGA

Sumário

Motivação

Range sum query

- O problema conhecido como *range sum query* ($RSQ(i, j)$) consiste em determinar a soma de todos os elementos de uma sequência a_k cujos índices pertencem ao intervalo $[i, j]$
- Por exemplo, se $a_k = \{1, 2, 3, 4, 5\}$, então $RSQ(2, 4) = 9, RSQ(3, 3) = 3$ e $RSQ(1, 5) = 15$
- Cada *query* pode ser respondida em $O(N)$, onde N é o número de elementos da sequência a_k, iterando em todos os elementos $a_i, a_{i+1}, \ldots, a_j$
- Se, para um mesmo vetor a_k, forem realizadas Q *queries*, a complexidade do algoritmo seria $O(NQ)$
- Contudo, esta complexidade pode ser reduzida para $O(N + Q)$, se o sequência for pré-processada, de modo que cada *query* pode ser respondida em $O(1)$

Implementação do RSQ com complexidade $O(N)$ por *query*

```cpp
template<typename T>
T RSQ(const vector<T>& as, int i, int j)
{
    T sum = 0;

    for (int k = i; k <= j; ++k)
        sum += a[k];

    return sum;
}
```

Soma dos prefixos de uma sequência

- Seja p_i a soma dos elementos do prefixo $a[1..i]$ da sequência a_k, isto é,

$$p_i = \sum_{k=1}^{i} a_k$$

- Por exemplo, para $a_k = \{1, 2, 3, 4, 5\}$, segue que $p_k = \{1, 3, 6, 10, 15\}$
- A sequência p_k, com $p_0 = 0$, pode ser utilizada para determinar $RSQ(i, j)$ em $O(1)$
- De fato,

$$RSQ(i, j) = p_j - p_{i-1}$$

- Como a sequência p_k é gerada com complexidade $O(N)$, a resposta para Q *queries* tem complexidade $O(N + Q)$

Implementação da soma dos prefixos

```cpp
// Os elementos as tem índices de 1 a N
template<typename T>
vector<T> prefix_sum(const vector<T>& as, int N)
{
    vector<T> ps(N + 1, 0);

    for (size_t i = 1; i <= N; ++i)
        ps[i] = ps[i - 1] + as[i];

    return ps;
}

template<typename T>
T RSQ(const vector<T>& ps, int i, int j)
{
    return ps[j] - ps[i - 1];
}
```

Range sum queries **com sequência dinâmica**

- Embora a soma dos prefixos resolva o problema quando a sequência a_k é estática, ela não se aplica em sequências dinâmicas
- Se a sequência modificar um ou mais de seus elementos entre duas *queries*, a sequência p_k também será modificada, e esta atualização tem complexidade $O(N)$, voltando à complexidade da solução original
- Para responder *range sum queries* em sequências dinâmicas com melhor complexidade é preciso o uso de estruturas mais sofisticadas, que permitam a atualização das somas pré-computadas de forma eficiente
- Uma destas estruturas é a *Binary Indexed Tree* (*BITree* ou *Fenwick Tree*), proposta pelo professor Peter M. Fenwick em 1994

Definição

Árvore de Fenwick

- A árvore de Fenwick (*Binary Indexed Tree*) é uma estrutura de dados que permite responder *range sum queries* de forma eficiente
- Ela suporta as operações de *range sum query* e a atualização de valores da sequência com complexidade $O(\log N)$
- A ideia principal da árvore de Fenwick é armazenar as somas de certos intervalos de índices da sequência de modo que qualquer intervalo $[1, j]$ possa ser representado, de forma única, pela união de, no máximo, $O(\log N)$ intervalos disjuntos
- Embora ela seja uma árvore, ela é implementada implicitamente por meio de um *array*, de forma semelhante à utilizada pelas pelas *heaps* binárias

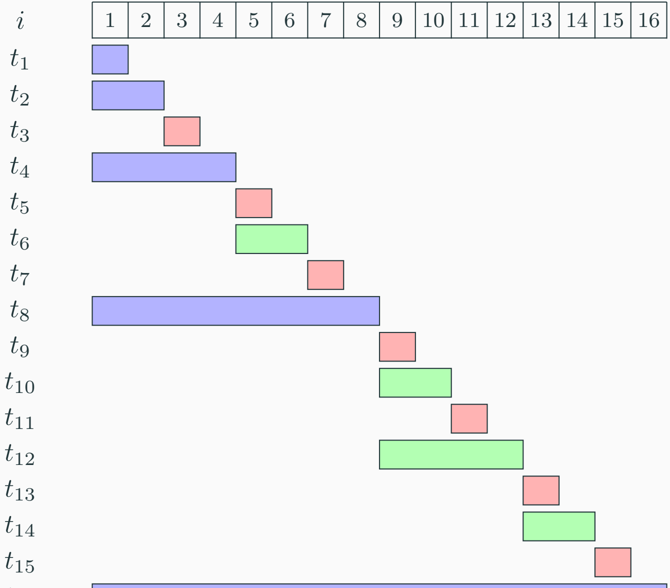
Implementação de uma árvore de Fenwick

- Assim como as *heaps* binárias, o índice zero do *array* t é descartado
- Seja $p(n)$ a maior potência de 2 que divide o inteiro positivo n
- Assim, t_i será a soma dos elementos de a_k cujos índices pertencem ao intervalo $I_i = [i - p(i) + 1, i]$, isto é,

$$t_i = \sum_{k=i-p(i)+1}^{i} a_k$$

- Por exemplo, para $a_k = \{1, 2, 3, 4, 5, 6, 7, 8\}$, segue que $t_k = \{1, 3, 3, 10, 5, 11, 7, 36\}$
- Esta escolha de intervalos permite a representação de qualquer intervalo $[1, j]$ por meio da união de $O(\log N)$ intervalos disjuntos cujas somas estão armazenadas no *array* t_k

Visualização dos intervalos correspondentes aos elementos t_i



Exemplo de representação de uma BITree em C++

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

using namespace std;

template<typename T>
class BITree {
private:
    vector<T> ts;
    size_t N;

public:
    BITree(size_t n) : ts(n + 1, 0), N(n) {}
```

Range sum queries

Range sum query em uma árvore de Fenwick

- Considere que
$$[1, j] = I_{k_1} \cup I_{k_2} \cup \ldots I_{k_r},$$
com $r = O(\log N)$, $I_i \cap I_j = \emptyset$ se $i \neq j$ e I_k é o intervalo associado a t_k
- Por exemplo,
$$[1, 15] = [1, 8] \cup [9, 12] \cup [13, 14] \cup [15, 15]$$
- Defina
$$S(j) = \sum_{i=k_1}^{k_r} t_i$$
- Assim, fazendo $t_0 = 0$, segue que
$$RSQ(i, j) = S(j) - S(i-1)$$

Identificação da decomposição de intervalos

- Para encontrar a decomposição de um intervalo $[1, j]$ em intervalos disjuntos associados aos elementos t_k, é preciso computar os valores de $p(n)$
- De fato, $p(n)$ corresponde ao *bit* menos significativo de n em base binária
- Este *bit* pode ser determinado de forma eficiente através de uma operação binária

$$p(n) = n \wedge (-n),$$

  onde $\wedge$ corresponde ao "e" lógico
- Os índices r_i dos intervalos I que decompõem $[1, j]$ formam a sequência de inteiros positivos

$$j, j - p(j), [j - p(j)] - p(j - p(j)), \ldots,$$

Implementação da RSQ em uma BITree

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

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

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

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

        return sum;
    }
```

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k								
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2							
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2						
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1					

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6				

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1			

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1	4		
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1	4	-2	
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = RSQ(1, 7) - RSQ(1, 2)$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = \mathbf{RSQ(1, 7)} - RSQ(1, 2)$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

	1	2	3	4	5	6	7	8
t_k	2	2	1	6	-1	4	-2	12
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = \mathbf{8} - RSQ(1, 2)$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3,7)$

	1	2	3	4	5	6	7	8
t_k	2	2	1	6	-1	4	-2	12
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3,7) = 8 - \mathbf{RSQ(1,2)}$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

	1	2	3	4	5	6	7	8
t_k	2	2	1	6	-1	4	-2	12
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = 8 - \mathbf{2}$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$RSQ(3, 7) = 6$

Construção e atualização de uma BITree

Atualização de um elemento

- A árvore de Fenwick permite a atualização de um valor da sequência a_k original
- Para incrementar o elemento a_i em x unidades, é preciso adicionar x em todos os intervalos onde este elemento é contabilizado
- A sequência dos intervalos $I_{k_1}, I_{k_2}, \ldots, I_{k_s}$, com $s = O(\log N)$, que contabilizam a_i, é tal que $k_1 = i$ e

$$k_j = k_{j-1} + p(k_{j-1})$$

- A sequência é interrompida em k_s, onde $k_{s+1} > N$
- Assim, a operação `add(i, x)` tem complexidade $O(\log N)$

Visualização de uma atualização em uma BITree

Atualização: $a_3 = a_3 + 4$

	1	2	3	4	5	6	7	8
t_k	2	2	1	6	-1	4	-2	12
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma atualização em uma BITree

Atualização: $a_3 = a_3 + 4$

t_k	2	2	5	6	-1	4	-2	12

	1	2	3	4	5	6	7	8
a_k	2	0	5	3	-1	5	-2	4

$r_1 = 3$

Visualização de uma atualização em uma BITree

Atualização: $a_3 = a_3 + 4$

t_k	2	2	5	10	-1	4	-2	12

	1	2	3	4	5	6	7	8
a_k	2	0	5	3	-1	5	-2	4

$$r_2 = r_1 + p(r_1) = 3 + 1 = 4$$

Visualização de uma atualização em uma BITree

Atualização: $a_3 = a_3 + 4$

t_k	2	2	5	10	-1	4	-2	17

	1	2	3	4	5	6	7	8
a_k	2	0	5	3	-1	5	-2	4

$$r_3 = r_2 + p(r_2) = 4 + 4 = 8$$

Implementação do método add(i, x)

```
public:
    void add(size_t i, const T& x)
    {
        if (i == 0)
            return;

        while (i <= N)
        {
            ts[i] += x;
            i += LSB(i);
        }
    }
};
```

Construção de uma árvore de Fenwick

- Uma maneira de se construir uma árvore de Fenwick é começar com um vetor cujos elementos são todos iguais a zero
- Depois, cada valor a ser atribuído à i-ésima posição deve ser somado a esta posição
- Como a operação de atualização/soma tem complexidade $O(\log N)$, onde N é o número máximo de elementos a serem inseridos na árvore, esta rotina tem complexidade $O(N \log N)$

Visualização da construção de uma BITree

Atualização: $a_1 = a_1 + 2$

t_k	0	0	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização da construção de uma BITree

Atualização: $a_1 = a_1 + 2$

t_k	2	0	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$r_1 = 1$

Visualização da construção de uma BITree

Atualização: $a_1 = a_1 + 2$

t_k	2	2	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_2 = r_1 + p(r_1) = 1 + 1 = 2$$

Visualização da construção de uma BITree

Atualização: $a_1 = a_1 + 2$

t_k	2	2	0	2	0	0	0	0
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_3 = r_2 + p(r_2) = 2 + 2 = 4$$

Visualização da construção de uma BITree

Atualização: $a_1 = a_1 + 2$

	1	2	3	4	5	6	7	8
t_k	2	2	0	2	0	0	0	2
a_k	2	0	1	3	-1	5	-2	4

$$r_4 = r_3 + p(r_3) = 4 + 4 = 8$$

Visualização da construção de uma BITree

Atualização: $a_2 = a_2 + 0$

t_k	2	2	0	2	0	0	0	2

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Nada a fazer

Visualização da construção de uma BITree

Atualização: $a_3 = a_3 + 1$

		1	2	3	4	5	6	7	8
t_k		2	2	1	2	0	0	0	2
a_k		2	0	1	3	-1	5	-2	4

$r_1 = 3$

Visualização da construção de uma BITree

Atualização: $a_3 = a_3 + 1$

t_k								
	2	2	1	3	0	0	0	2

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_2 = r_1 + p(r_1) = 3 + 1 = 4$$

Visualização da construção de uma BITree

Atualização: $a_3 = a_3 + 1$

	1	2	3	4	5	6	7	8
t_k	2	2	1	3	0	0	0	3
a_k	2	0	1	3	-1	5	-2	4

$$r_3 = r_2 + p(r_2) = 4 + 4 = 8$$

Visualização da construção de uma BITree

Atualização: $a_4 = a_4 + 3$

t_k	2	2	1	6	0	0	0	3
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$r_1 = 4$

Visualização da construção de uma BITree

Atualização: $a_4 = a_4 + 3$

t_k	2	2	1	6	0	0	0	6
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_2 = r_1 + p(r_1) = 4 + 4 = 8$$

Visualização da construção de uma BITree

Atualização: $a_5 = a_5 - 1$

t_k	2	2	1	6	-1	0	0	6

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$r_1 = 5$

Visualização da construção de uma BITree

Atualização: $a_5 = a_5 - 1$

t_k	2	2	1	6	-1	-1	0	6

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_2 = r_1 + p(r_1) = 5 + 1 = 6$$

Visualização da construção de uma BITree

Atualização: $a_5 = a_5 - 1$

t_k	2	2	1	6	-1	-1	0	5

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_3 = r_2 + p(r_2) = 6 + 2 = 8$$

Visualização da construção de uma BITree

Atualização: $a_6 = a_6 + 5$

t_k	2	2	1	6	-1	4	0	5
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$r_1 = 6$

Visualização da construção de uma BITree

Atualização: $a_6 = a_6 + 5$

t_k	2	2	1	6	-1	4	0	10

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_2 = r_1 + p(r_1) = 6 + 2 = 8$$

Atualização: $a_7 = a_7 - 2$

	1	2	3	4	5	6	7	8
t_k	2	2	1	6	-1	4	-2	10
a_k	2	0	1	3	-1	5	-2	4

$r_1 = 7$

Visualização da construção de uma BITree

Atualização: $a_7 = a_7 - 2$

t_k	2	2	1	6	-1	4	-2	8

	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$r_2 = r_1 + p(r_1) = 7 + 1 = 8$$

Visualização da construção de uma BITree

Atualização: $a_8 = a_8 + 4$

	1	2	3	4	5	6	7	8
t_k	2	2	1	6	-1	4	-2	12
a_k	2	0	1	3	-1	5	-2	4

$r_1 = 8$

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. HackerEarth. Fenwick (Binary Indexed) Trees, acesso em 06/05/2019.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2010.
4. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, acesso aberto[1].
5. Wikipedia. Fenwick Tree, acesso em 06/05/2019.

[1] https://cses.fi/book/index.html