

Aula 6 · Busca Binária, Divisão e Conquista e Árvores de Fenwick

Desafios de Programação

Fernando Kiotheka Vinícius Tikara Date

UFPR

26/04/2023

Busca Binária

Esse tal de lg parece bom!

- Os logaritmos aparecem toda hora nos algoritmos.
- Geralmente isso advém de resolver problemas por meio da divisão em subproblemas menores.
- O número de vezes que a gente pode dividir um problema sucessivamente por dois é dado por $\log_2$ ou lg.
- Se a gente dividir por três acabamos com $\log_3$, mas também podemos fazer operações no caminho dessa recursão, acabando com complexidades mais complicadas. Isso é tópico da disciplina de Análise de Algoritmos.
- Já vimos algoritmos e estruturas logarítmicas: A exponenciação binária, e as árvores rubro-negras (set e map) têm essa complexidade embutida no seu funcionamento.
- O algoritmo que mais encapsula esse comportamento em sua essência é a *busca binária*.

A busca binária

Em Algoritmos I devem te contar a historinha de como você acharia uma palavra no dicionário. Como o dicionário está ordenado, um jeito bem eficiente é abrir o meio, verificar se o que você está procurando está antes ou depois, e ir dividindo o que resta ao meio até encontrar a palavra que você quer. Isso é eficiente, pois é logarítmico, e $\lg$ é uma operação eficiente:

n	$\lg n$	n	$\lg n$	n	$\lg n$
10^0	0,000	10^{10}	33,219	10^{20}	66,438
10^1	3,321	10^{11}	36,541	10^{21}	69,760
10^2	6,643	10^{12}	39,863	10^{22}	73,082
10^3	9,965	10^{13}	43,185	10^{23}	76,404
10^4	13,287	10^{14}	46,506	10^{24}	79,726
10^5	16,609	10^{15}	49,828	10^{25}	83,048
10^6	19,931	10^{16}	53,150	10^{26}	86,370
10^7	23,253	10^{17}	56,472	10^{27}	89,692
10^8	26,575	10^{18}	59,794	10^{28}	93,013
10^9	29,897	10^{19}	63,116	10^{29}	96,335

O que precisa pra fazer busca binária?

- É preciso estabelecer um **predicado** de busca.
- Ele particiona o espaço de busca em duas regiões contíguas.

Por exemplo, $P(i) = E_n[i] \leq 2$ particiona um espaço ordenado

$$E_n = \left[\begin{array}{cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \right]$$

em duas regiões, compostas por valores de verdadeiro ou falso:

$$E_b = \left[\begin{array}{cccccccc} & T & T & F & F & F & F & F \\ 0 & T & 1 & T & 2 & F & 3 & F & 4 & F & 5 & F & 6 & F & 7 \end{array} \right]$$

Quando você consegue fazer isso, a busca binária vai te permitir achar exatamente a posição onde o espaço de busca é particionado em $\lg n$ (sendo n o tamanho do espaço de busca).

Note o espaço!

- Nem todo espaço de busca é adequado.
- O algoritmo tem que saber **pra onde ir** em toda iteração.

Por exemplo, imagine que a gente tem uma sequência de bits da seguinte forma: $0^*1^*0^*$. É possível encontrar o primeiro bit 1 usando busca binária? De forma geral, não.

- Se encontrarmos um 1 por acidente, sabemos para onde ir! (para a esquerda)
- Porém, se sempre encontrarmos um 0, não sabemos para que lado ir, pois a sequência de 1s pode estar à direita ou à esquerda.

Assim, no pior caso, achar o primeiro bit 1 nessa sequência é $\mathcal{O}(n)$. A busca binária não nos ajuda a melhorar essa complexidade. Esse problema se resume ao problema de achar a posição de um valor em um vetor ordenado deslocado com elementos duplicados.

Mas no que eu posso aplicar, então?

- Posição à esquerda ou à direita de valor em vetor ordenado.
- Contagem de ocorrências de valor em vetor ordenado.
- Posição de valor em vetor ordenado deslocado sem duplicados.
- Posição de máximo/mínimo de função unimodal.
- Raiz quadrada de um número (método de bisecção).
- i -ésimo elemento de união de vetores ordenados.
- Elemento em vetor ordenado que é igual ao seu índice.
- Muito mais, inclusive envolvendo números em ponto flutuante.

Achando posição depois do valor em vetor ordenado

Código upperboundmanual.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; while (cin >> n) {
        vector<int> v (n); for (int& x : v) { cin >> x; }
        int lo = 0, hi = n;
        while (lo < hi) {
            int mi = lo + (hi-lo) / 2; // evite overflow!
            if (v[mi] <= 2) { lo = mi + 1; }
            else { hi = mi; }
        } cout << lo << "\n";
    }
}
```

Entrada	Saída
7 1 2 3 4 5 6 7	2
7 1 2 2 2 3 4 5	4
7 1 1 1 2 2 3 4	5

Achando posição depois do valor em vetor ordenado (STL)

Código upperbound.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; while (cin >> n) {
        vector<int> v (n); for (int& x : v) { cin >> x; }
        auto it = upper_bound(v.begin(), v.end(), 2);
        cout << it - v.begin() << "\n";
    }
}
```

Entrada	Saída
7 1 2 3 4 5 6 7	2
7 1 2 2 2 3 4 5	4
7 1 1 1 2 2 3 4	5

Operações prontas na STL

- `upper_bound(iterator begin, end, T val)`: Acha posição depois do valor em vetor ordenado ($P(i) = v[i] \leq v$).
- `lower_bound(iterator begin, end, T val)`: Acha posição do primeiro valor em vetor ordenado ($P(i) = v[i] < v$).
- `binary_search(iterator begin, end, T val)`: Verifica se o valor está no vetor ordenado (utilizando `lower_bound`).

Usando essas operações é possível calcular a quantidade de números em um vetor ordenado (`upper_bound - lower_bound`).

Qualquer outra coisa: Você vai ter que implementar na mão.

Principalmente aquelas que o valor é computado por alguma função, e não tem todos os valores prontinhos em um vetor (pré-calculá-los todos seria $\mathcal{O}(n)$ ou pior).

lower_bound nas estruturas da STL

Quando falamos das estruturas da STL, deixamos de fora essas funções, mas todas as estruturas ordenadas, como map, set, multiset, etc tem funções **próprias** de lower_bound e upper_bound.

- Preste muita atenção que é `s.lower_bound(v)` e não `lower_bound(s.begin(), s.end(), v)`.
- A complexidade do lower_bound com iteradores vai se reduzir a $\mathcal{O}(n)$ se os iteradores não suportarem acesso aleatório (o que é o caso para essas estruturas).
- Isso é muito interessante para implementar várias coisas legais com essas estruturas (por exemplo um diagrama de Voronoi).
- Essa técnica pode ser aplicada em qualquer coisa ordenada. É possível por exemplo fazer um lower_bound para a Árvore de Fenwick que veremos daqui a pouco.

Quero ler mais sobre!

- Temos um material específico só sobre busca binária: <https://www.inf.ufpr.br/maratona/nlgn/busca-binaria/>.
- Os detalhes do algoritmo estão mais bem explicadinhos e há mais alguns exemplos e diversas implementações que podem ser mais intuitivas.
- Um exemplo é o que vou mostrar a seguir: Passo binário é um jeito de “compactar” a busca binária, e permite que várias buscas binárias possam funcionar simultaneamente (do outro jeito você precisaria de duas buscas, uma seguida da outra).

Implementação alternativa: Passo binário

Código binarystep.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; while (cin >> n) {
        vector<int> v (n); for (int& x : v) { cin >> x; }
        int l = n, r = n;
        for (int p = n; p >= 1; p /= 2) {
            while (l-p >= 0 && !(v[l-p] < 2)) { l -= p; }
            while (r-p >= 0 && !(v[r-p] <= 2)) { r -= p; }
        }
        cout << r - l << "\n";
    }
}
```

Entrada	Saída
7 1 2 3 4 5 6 7	1
7 1 2 2 2 3 4 5	3
7 1 1 1 2 2 3 4	2

Divisão e Conquista

Divisão e Conquista

- Algoritmos que implementam os passos:
 - **Dividir:** Quebrar o problema em pedaços.
 - **Conquistar:** Resolver o problema de forma recursiva.
 - **Combinar:** Combinar as subsoluções em uma solução.
- Complexidade pelo Teorema Mestre (Análise de Algoritmos).
- Exemplo prático: Contar inversões usando o Merge Sort.

Contando inversões

Inversões... que?

- Em ciência da computação e em matemática discreta, uma sequência tem uma inversão quando dois de seus elementos estão fora de sua ordem natural.
- Seja A uma sequência. Se $i < j$ e $A[i] > A[j]$, então o par de índices (i, j) ou o par de elementos $(A[i], A[j])$ é chamado de **inversão** de A .

Notação: I_v^0 é um conjunto de inversão de par de índices indexados em 0 do vetor v . I_v é o conjunto de inversão de elementos de v .

$$v_1 = \left[\begin{smallmatrix} 0 & 4 & 1 & 2 & 1 & 3 & 4 \end{smallmatrix} \right]$$

$$I_{v_1}^0 = \{(0, 1), (0, 2), (0, 3), (1, 2)\}$$

$$I_{v_1} = \{(4, 2), (4, 1), (4, 3), (2, 1)\}$$

Exemplo de inversão

$$v_1 = \left[\begin{array}{cccccc} & & & & & \\ 0 & 1 & 3 & 5 & 7 & \end{array} \right]$$
$$I_{v_1} = \{\}$$

Exemplo de inversão

$$v_1 = \begin{bmatrix} 0 & 1 & 3 & 5 & 7 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

Exemplo de inversão

$$v_2 = \left[\begin{smallmatrix} 0 & 1 & 3 & 7 & 5 \\ 0 & 1 & 3 & 7 & 5 \end{smallmatrix} \right]$$
$$I_{v_2} = \{(2, 3)\}$$

Exemplo de inversão

$$v_2 = \left[\begin{smallmatrix} 0 & 1 & 3 & 7 & 5 \\ 0 & 1 & 2 & 3 & 4 \end{smallmatrix} \right]$$



$$I_{v_2} = \{(2, 3)\}$$

Exemplo de inversão

$$v_3 = \left[\begin{array}{cccccc} & & & & & \\ & & & & & \\ 0 & 1 & 7 & 3 & 5 & \end{array} \right]$$

$$I_{v_3} = \{(1, 2), (1, 3)\}$$

Exemplo de inversão

$$v_3 = \left[\begin{array}{c} \\ 0 \end{array} \begin{array}{c} 1 \\ 1 \end{array} \begin{array}{c} 7 \\ 2 \end{array} \begin{array}{c} 3 \\ 3 \end{array} \begin{array}{c} 5 \\ 4 \end{array} \right]$$



$$I_{v_3} = \{(1, 2), (1, 3)\}$$

Exemplo de inversão

$$v_4 = \begin{bmatrix} 0 & 7 & 1 & 3 & 5 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$
$$I_{v_4} = \{(0, 1), (0, 2), (0, 3)\}$$

Exemplo de inversão

$$v_4 = \left[\begin{smallmatrix} 0 & 7 & 1 & 3 & 5 \\ & 1 & 2 & 3 & 4 \end{smallmatrix} \right]$$



$$I_{v_4} = \{(0, 1), (0, 2), (0, 3)\}$$

Exemplo de inversão

$$v_5 = \left[\begin{smallmatrix} 0 & 7 & 1 & 5 & 3 \\ & 1 & 2 & 3 & 4 \end{smallmatrix} \right]$$

$$I_{v_5} = \{(0, 1), (0, 2), (0, 3), (2, 3)\}$$

Exemplo de inversão

$$v_5 = \left[\begin{smallmatrix} & & & & \\ 0 & 7 & 1 & 5 & 3 \\ & 1 & 2 & 3 & 4 \end{smallmatrix} \right]$$


$$I_{v_5} = \{(0, 1), (0, 2), (0, 3), (2, 3)\}$$

Exemplo de inversão

$$v_6 = \left[\begin{smallmatrix} 0 & 7 & 1 & 5 & 2 & 1 & 3 & 4 \end{smallmatrix} \right]$$

$$I_{v_6} = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)\}$$

Exemplo de inversão

$$v_6 = \left[\begin{smallmatrix} & 0 & 7 & 1 & 5 & 2 & 1 & 3 & 4 \end{smallmatrix} \right]$$


$$I_{v_6} = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)\}$$

Exemplo de inversão

$$v_7 = \left[\begin{smallmatrix} & 7 & 5 & 3 & 1 \\ 0 & 1 & 2 & 3 & 4 \end{smallmatrix} \right]$$

$$I_{v_7} = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$$

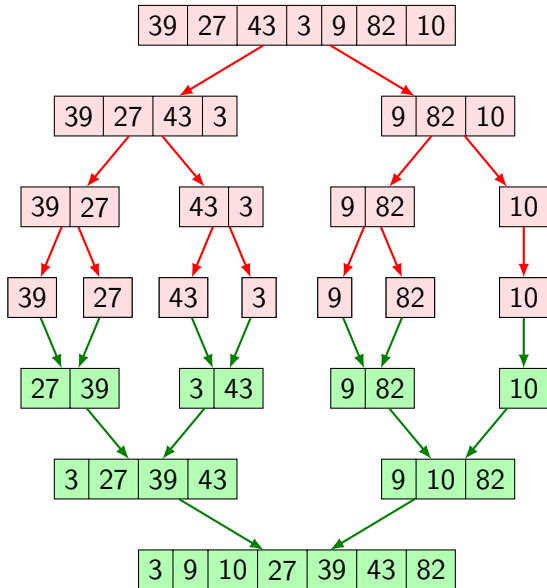
Exemplo de inversão

$$v_7 = \left[\begin{smallmatrix} 0 & 7 & 1 & 5 & 2 & 3 & 4 \end{smallmatrix} \right]$$

$$I_{v_7} = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$$

$$|I_{v_7}| = n(n-1)/2 = (4 \cdot 3)/2 = 6$$

Merge Sort



Merge Sort

Código mergesort.h

```
int swaps = 0;
void merge_sort(int l, int r) {
    if (r - l == 1) { return; }
    int mi = l + (r - l) / 2;
    merge_sort(l, mi); merge_sort(mi, r);
    vector<int> aux (r - l);
    int i = l, j = mi;
    for (int k = 0; k < r - l; k++) {
        if (i < mi && j < r) {
            if (!(a[i] < a[j])) { swaps += mi - i; }
            if (a[i] < a[j]) { aux[k] = a[i++]; }
            else { aux[k] = a[j++]; }
        } else if (i < mi) { aux[k] = a[i++]; }
        else { aux[k] = a[j++]; }
    }
    copy(aux.begin(), aux.end(), a.begin()+l);
}
```


Fusão

$$v_1 = \begin{bmatrix} \overset{i}{\downarrow} 2 & 3 & 5 & \overset{j}{\downarrow} 1 & 4 & 6 \end{bmatrix}$$

$swaps = 0$

Fusão

$$v_1 = \begin{bmatrix} \overset{i}{\downarrow} 2 & 3 & 5 & \overset{j'}{\downarrow} 1 & \overset{j}{\downarrow} 4 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

$swaps = 0$

Fusão

$$v_2 = \begin{bmatrix} \overset{i}{\downarrow} 2 & 3 & \overset{j'}{\downarrow} 1 & 5 & \overset{j}{\downarrow} 4 & 6 \end{bmatrix}$$

$swaps = 1$

Fusão

$$v_3 = \begin{bmatrix} \overset{i}{\downarrow} 2 & \overset{j'}{\downarrow} 1 & 3 & 5 & \overset{j}{\downarrow} 4 & 6 \end{bmatrix}$$

$swaps = 2$

Fusão

$$v_4 = \begin{bmatrix} \overset{j'}{\downarrow} 1 & \overset{i}{\downarrow} 2 & 3 & 5 & \overset{j}{\downarrow} 4 & 6 \end{bmatrix}$$

$swaps = 3$

Fusão

$$v_4 = \left[\begin{array}{cccccc} & & i & & j & \\ & & \downarrow & & \downarrow & \\ 0 & 1 & 2 & 3 & 5 & 4 & 6 \end{array} \right]$$

$swaps = 3$

Fusão

$$v_4 = \left[\begin{array}{c} \\ 1 2 3 5 4 6 \\ \end{array} \right]$$

i j
↓ ↓

$$swaps = 3$$

Fusão

$$v_4 = \left[\begin{array}{cccccc} & & & & i & j \\ & & & & \downarrow & \downarrow \\ 0 & 1 & 2 & 3 & 5 & 4 & 6 \end{array} \right]$$

$swaps = 3$

Fusão

$$v_4 = \begin{bmatrix} 0 & 1 & 2 & 3 & 5 & 4 & 6 \end{bmatrix}$$

i j' j
↓ ↓ ↓
 5_4 4_5 6_6

$swaps = 3$

Fusão

$$v_5 = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

j' i j
↓ ↓ ↓

$swaps = 4$

Implementação de contar inversões

Código mergesort.cpp

```
#include <bits/stdc++.h>
using namespace std;
vector<int> a (1e5+15);
#include "mergesort.h"
int main() {
    int n; while (cin >> n) {
        for (int i = 0; i < n; i++)
            cin >> a[i];
        swaps = 0;
        merge_sort(0, n);
        cout << swaps << "\n";
    }
}
```

Entrada	Saída
4 4 2 1 3	4
4 7 5 3 1	6
6 2 3 5 1 4 6	4

Quero saber mais sobre inversões!

Na oficina do Clube de Programação de 9 de março de 2021, falamos mais sobre inversões (<http://cdp.dainf.ct.utfpr.edu.br/history.html#of9mar2021>) e como dá pra resolver com:

- Merge Sort, o que acabamos de ver.
- Árvore de Fenwick que veremos agora!
- Árvore de Segmentos que também veremos a seguir.

A complexidade de todas as soluções é $\mathcal{O}(n \lg n)$.

Árvores

Somando intervalos

Dado um vetor v e os limites $[l, r]$, devolva a soma $\sum_{i=l}^r v_i$

- $\mathcal{O}(n)$ para calcular a soma uma vez
- Com consultas, fica $\mathcal{O}(qn)$
- Soma acumulada (aula passada) permite consultas $\mathcal{O}(1)$ — conseguimos $\mathcal{O}(q)$!

E se fosse permitido alterar os valores no meio do problema?

Somando intervalos e alterando valores

Dado um vetor v , $|v| = n$, responda a q consultas, que podem ser:

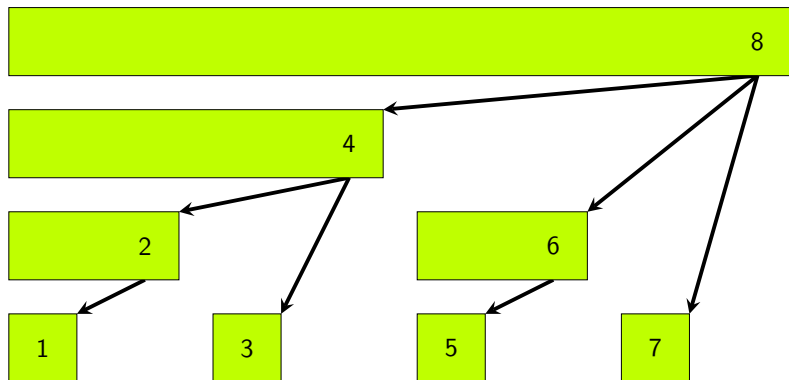
- Dados os limites $1 \leq l \leq r \leq n$, devolva a soma $\sum_{i=l}^r v_i$
- Dados os valores $1 \leq i \leq n$ e x , atribua a v_i o valor x

Agora precisaríamos atualizar a soma de prefixos toda vez, e voltamos ao $\mathcal{O}(qn)$.

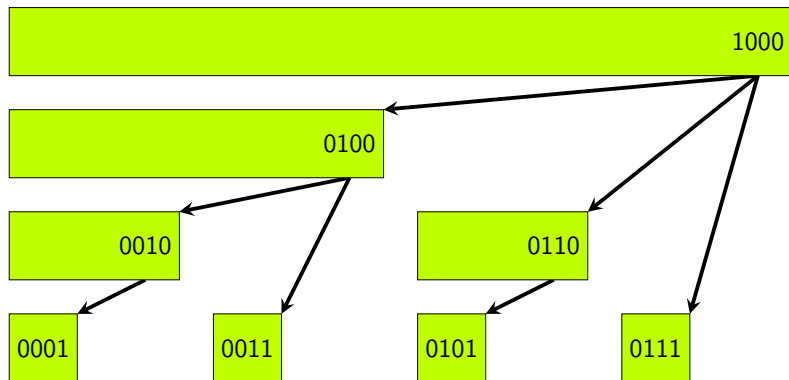
Árvore de Fenwick (BIT)

- Também conhecida como Binary Indexed Tree.
- Percorre o vetor em $\mathcal{O}(\lg n)$.
- Responde a problemas de consulta e alteração com funções reversíveis.
- É essencialmente um super vetor de prefixos.
- Cada posição da árvore guarda o valor para alguns elementos do vetor.
- Quais?

Responsabilidades de cada posição da árvore



Responsabilidades de cada posição da árvore (em binário)



O caminho do bit menos significativo

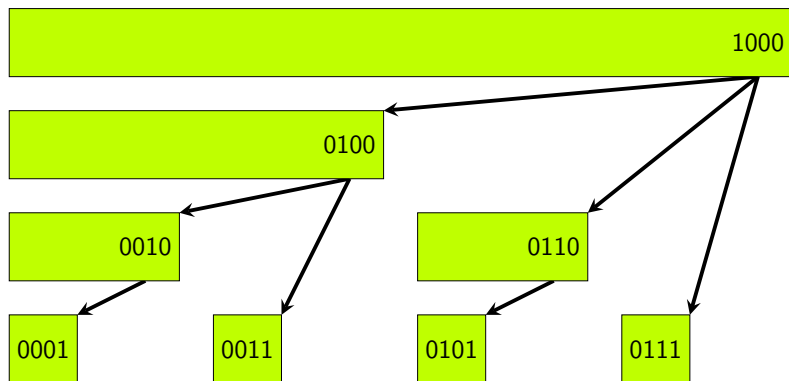
Para cada índice $1 \leq i \leq n$, vamos guardar a soma dos últimos $\text{LSOne}(i)$ elementos (incluindo i), onde $\text{LSOne}(x)$ devolve o valor do bit menos significativo de x .

Observações importantes:

- **Caminho do decremento:** Para qualquer valor de i , podemos decrementá-lo em $\text{LSOne}(i)$ e eventualmente chegaremos em $i = 0$: $111 \rightarrow 110 \rightarrow 100 \rightarrow 000$.
- **Caminho do incremento:** Para qualquer valor de $i > 0$, podemos incrementá-lo em $\text{LSOne}(i)$ e eventualmente teremos $i \geq n$: $001 \rightarrow 010 \rightarrow 100 \rightarrow \dots$.
- Seja $x = \text{LSOne}(i)$. Os $x - 1$ valores imediatamente antes de i têm i em seus caminhos do incremento. Para $i = 6$, temos 0110 e 0101 .
- O caminho do decremento de i passa por um valor no caminho do incremento de cada valor $j \leq i$.

Essa última fica mais fácil com um desenho.

Caminhando com LSONe(i)



Associação de elementos na BIT

Nossa BIT é um vetor de tamanho $n + 1$ (ignoramos o 0). Vamos atribuir a cada índice i a responsabilidade de guardar a soma no intervalo $[i - \text{LSOne}(i) + 1, i]$.

- O índice i guarda o valor de consulta para todos os elementos que ele “cobre”.
- **Consulta:** Se quisermos a soma de 1 até i , somamos os valores salvos no caminho do decremento de i ($\mathcal{O}(\log n)$).
- Se quisermos a soma de l até r , fazemos a operação anterior duas vezes.
- **Atualização:** Se mudarmos um valor no índice j , seguimos o caminho do incremento de j para atualizar os valores que dependem dele ($\mathcal{O}(\log n)$).

Implementação da BIT

Código bit.h

```
vector<int> bit (N, 0);  
void add(int i, int delta) {  
    for (; i < N; i += i & (-i))  
        bit[i] += delta;  
}  
int get(int i) {  
    int ans = 0;  
    for (; i > 0; i -= i & (-i))  
        ans += bit[i];  
    return ans;  
}
```

Usando a Árvore de Fenwick

Mais do que entender, precisamos saber usar. Geralmente, podemos usar a BIT das seguintes formas:

- Atualização em posição e consulta de intervalo.
 - Alteração da posição i é feita usando `add(i, delta)`.
 - Consulta do intervalo $[a, b]$ é feita fazendo `get(b) - get(a-1)`.
- Atualização em intervalo e consulta em posição.
 - Alteração do intervalo $[a, b]$ é feita fazendo `add(a, delta)` e `add(b+1, -delta)`.
 - Consulta da posição i é feita fazendo `get(i)`.

Outros usos possíveis envolvem o uso de árvores BIT 2D, e existem outros hacks utilizando de várias árvores para por exemplo permitir a alteração e consulta em intervalos, veja mais em https://cp-algorithms.com/data_structures/fenwick.html.

Atualização em posição e consulta em intervalo

Código updatepos.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15;
#include "bit.h"
int main() {
    char op; int a, b, i, d; string s;
    while (cin >> op) {
        if (op == '+') {
            cin >> i >> d; add(i, d);
        } else if (op == 'q') {
            cin >> a >> b;
            cout << get(b) - get(a-1) << "\n";
        } else getline(cin, s);
    }
}
```


Atualização em intervalo e consulta em posição (cont.)

Entrada	Saída
# 0 0 0 0 0	3
+ 1 +1	6
# 1 0 0 0 0	2
+ 2 +2	-1
# 1 2 0 0 0	2
+ 5 +3	-3
# 1 2 0 0 3	
q 1 3	
q 1 5	
+ 2 -1	
# 1 1 0 0 3	
+ 3 -3	
# 1 1 -3 0 3	
q 1 2	
q 1 3	
q 1 5	
q 3 3	

Atualização em intervalo e consulta em posição

Código updateinterval.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15;
#include "bit.h"
int main() {
    char op; int a, b, d, i; string s;
    while (cin >> op) {
        if (op == '+') {
            cin >> a >> b >> d;
            add(a, d); add(b+1, -d);
        } else if (op == 'q') {
            cin >> i; cout << get(i) << "\n";
        } else getline(cin, s);
    }
}
```

Atualização em intervalo e consulta em posição (cont.)

Entrada	Saída
# 0 0 0 0 0	2
+ 1 5 +2	0
# 2 2 2 2 2	5
+ 4 5 +3	7
# 2 2 2 5 5	2
+ 1 2 -2	
# 0 0 2 5 5	
q 3	
q 2	
q 4	
+ 1 4 +2	
# 2 2 4 7 5	
q 4	
q 2	