MAC0345 - Desafios 2
Editorial Lista 1

Nathan Luiz

April 12, 2023

A. Range Queries and Copies .
Tópicos: Segtree Persistente

Este problema é uma aplicação direta de segtree persistente. Uma observação legal é que na terceira operação não é necessário criar um nó novo, apenas colocamos a versão atual com o mesmo id da versão anterior. Isso funciona pois os nós nunca são apagados e nem modificados.

B. The Untended Antiquity .
Tópicos: Segtree 2D

O problema basicamente adiciona e remove barreiras, que são retângulos em um grid, e quer saber se 2 pontos estão na mesma componente. Nós não aprendemos segtree 2D com lazy propagation, então somente podemos fazer updates pontuais. Observe que as componentes não se intersectam, mas podem ficar contidas umas nas outras. Vamos fazer uma seg2D que guarda o Xor do range. Essa solução será probabilística. Iremos atualizar de maneira que dois pontos na mesma componente tenham o mesmo valor na query e se tiverem em componentes distintas, o valor dá diferente com probabilidade alta. Suponha que o update é no retângulo $(x_1, y_1), (x_2, y_2)$, tal que $x_1 \leq x_2$ e $y_1 \leq y_2$. Seja R um número aleatório. Então, vamos fazer update em:

- update (x_1, y_1, R)
- update ($x_2 + 1$, y_1, R)
- update (x_1, $y_2 + 1$, R)
- update ($x_2 + 1$, $y_2 + 1$, R)

Temos $query(x, y)$ deve retornar a componente da casa (x, y) e é o Xor dos os valores no retângulo $(1, 1), (x, y)$ (prefixo da matriz). Observe que para todos os pontos dentro de uma mesma componente o Xor da query dá o mesmo. É necessário que R seja aleatório por causa de casos como o abaixo:

```
10 10 4
1 2 2 9 9
1 3 3 8 8
1 4 4 7 7
3 1 1 5 5
```

Se a área de fora tiver $R = 1$, a do meio $R = 2$ e a de dentro $R = 3$, o Xor de uma casa de dentro é 0, que é o mesmo de não estar dentro de nenhum retângulo.

A complexidade final é $O(Qlog^2N)$.

C. Polynomial Queries .

Tópicos: Lazy propagation

Vamos utilizar lazy propagation para resolver. As tags da lazy vão ser A e R, que são, respectivamente, o elemento inicial da PA e a razão. Essas tags guardam informação do nó de maneira que somamos A no elemento mais à esquerda, $A + R$ no segundo elemento, e por aí vai.

Como para somar duas $PA's$, basta somar o elemento inicial e a razão, então as transições podem ser feitas como abaixo:

```
struct Node {
    // nosso nó tem uma PA que soma
    // A no primeiro, A + R no segundo...
    int sum, A, R;
    Node () : sum (0), A (0), R (0) {}
    Node (int sum, int A, int R) : sum (sum), A (A), R (R) {}
    Node operator + (const Node &o) const {
        return Node (sum + o.sum, A + o.A, R + o.R);
    };
};

void push (int no, int ini, int fim) {
    if (node[no].R  != 0 || node[no].A != 0) {
        {
            int sz = fim - ini + 1;
            node[no].sum += sz * node[no].A + node[no].R * (sz - 1) * sz / 2;
        }

        if (ini != fim) {
            node[2 * no].R += node[no].R;
            node[2 * no].A += node[no].A;

            int meio = (ini + fim) / 2, sz_l = meio - ini + 1;
            node[2 * no + 1].R += node[no].R;
            node[2 * no + 1].A += node[no].A + node[no].R * sz_l;
        }

        node[no].A = 0;
        node[no].R = 0;
    }
}
```

A solução funciona e $O(NlogN)$.

D. Forest Queries II .

Tópicos: Segtree 2D

Este problema é aplicação direta de segtree 2D. Basta fazer uma seg de soma de maneira que cada casa é 1, se tiver árvore, e 0 caso contrário.

E. K-th Number .

Tópicos: Segtree persistente

Esse problema pode ser resolvido de algumas maneiras. Vou descrever uma utilizando segtree persistente em $O(NlogN)$.

Primeiramente, podemos comprimir coordenadas de maneira que cada número seja mapeado para um inteiro de 1 até n. Para cada posição do array, queremos uma versão que salva informação do prefixo inteiro em forma de segtree de frequência. Por exemplo, no array $A = [1, 3, 3, 5, 2, 4]$, a versão 3 do array é $f = [1, 0, 2, 0, 0, 0, ...]$, onde $f[i]$ é quantas vezes o elemento i ocorre no prefixo. Podemos montar uma versão para cada índice com seg persistente, pois versão $i + 1$ reaproveita a versão i.

Agora, dada uma lista de frequência, queremos achar o k-ésimo elemento desta lista. Como ela está guardada em uma seg, podemos fazer uma busca binária dentro da query! Suponha que estamos em um nó e queremos achar o k-ésimo elemento dele. Se o nó da esquerda tem pelo menos k elementos, vamos para esquerda. Caso contrário, supondo que o nó da esquerda tem L elementos, queremos achar o $(k - L)$-ésimo elemento do nó da direita.

A complexidade final é de $O(NlogN)$ memória e tempo.

F. Distinct Values Queries .
Tópicos: Segtree persistente

Vamos fazer uma segtree para cada posição que guarda informação do prefixo. Suponha que estamos analisando a segtree de versão i. Então, sendo j um índice qualquer:

- se $j > i$:
  $seg[j] = 0$.
- se $j \leq i$:
  $seg[j] = 1$ se j é a aparição mais à direita do elemento a_j no meu prefixo e 0 caso contrário.

Por exemplo, caso o array seja $a = [1, 1, 2, 1, 2, 3, 3, 3, 6]$, a segtree da versão 9 é $[0, 0, 0, 1, 1, 0, 0, 1, 1]$. Portanto, se quisermos saber a resposta de l até r, basta fazer uma query na versão r no intervalo de l até r. O update de uma versão para outra altera no máximo 2 casas da seg.

Observe que é possível responder as queries offline com esta solução, porém não há necessidade do uso de segtree persistente. A complexidade total é $O(NlogN + QlogN)$ em tempo e $O(NlogN)$ memoria.

G. Distinct Values Queries .
Tópicos: Lazy Propagation

Este problema é na minha opinião o mais difícil da lista pois os detalhes de implementação não são fáceis de codar. Mas a ideia é achar alguns parâmetros de lazy de maneira que fique parecido com o problema C.

Vamos utilizar os parâmetros F_l e F_r da lazy de um nó. Dessa forma,

- No 1^o elemento do nó temos F_l, F_r e queremos somar F_r;
- No 2^o elemento do nó temos F_r, $F_l + F_r$ e queremos somar $F_l + F_r$;
- No 3^o elemento do nó temos $F_l + F_r$, $F_l + 2F_r$ e queremos somar $F_l + 2F_r$;
- No 4^o elemento do nó temos $F_l + 2F_r$, $2F_l + 3F_r$ e queremos somar $2F_l + 3F_r$;
- No 5^o elemento do nó temos $2F_l + 3F_r$, $3F_l + 5F_r$ e queremos somar $3F_l + 5F_r$;
- No i-ésimo elemento do nó temos $f_{i-2}F_l + f_{i-1}F_r$, $f_{i-1}F_l + f_iF_r$ e queremos somar $f_{i-1}F_l + f_{i-1}F_r$;
- ...

Procuramos parâmetros que são parecidos com *Fibonacci* na forma em que são construídos. Experimente $F_l = 0, F_r = 1$. Tendo essa sequência, é necessário achar como fazer as transições. Vamos usar que $\sum_{i=1}^{r} f_i = f_{r+2} - 1$. É possível provar por indução. Recomendo pensar com calma nas transições, mas abaixo está um spoiler:

```
void push (int no, int ini, int fim) {
        if (node[no].f_l || node[no].f_r) {
            {
                int sz = fim - ini + 1;
                node[no].sum += sum_range (0, sz - 1) * node[no].f_l;
                node[no].sum += sum_range (1, sz) * node[no].f_r;
            }

            if (ini != fim) {
                node[2 * no].f_l += node[no].f_l;
                node[2 * no].f_r += node[no].f_r;

                int meio = (ini + fim) / 2, sz = meio - ini + 1;
                node[2 * no + 1].f_l += F[sz - 1] * node[no].f_l + F[sz] * node[no].f_r;
                node[2 * no + 1].f_r += F[sz] * node[no].f_l + F[sz + 1] * node[no].f_r;
            }

            node[no].f_l = node[no].f_r = 0;

        }
    }
```

A função `sum_range (l, r)` retorna $\sum_{i=l}^{r} f_i$.