Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Matheus de Mello Santos Oliveira

Árvores de Segmentos

São Paulo
25 de novembro de 2018

Árvores de Segmentos

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Carlos Eduardo Ferreira

São Paulo
25 de novembro de 2018

Resumo

A utilização da estrutura de dados conhecida como árvores de segmentos é bastante frequente em competições de programação. Este trabalho apresenta a teoria básica da estrutura e suas implementações assim como suas aplicações, extensões e ênfase na resolução de problemas de programação competitiva.

Palavras-chave: árvores de segmentos, implementação, aplicação, resolução de problemas, programação competitiva.

Sumário

1 Introdução **1**

2 A escolha do problema **3**

2.1 Motivação 3

2.2 Objetivo 4

2.3 Aplicação 4

2.4 Exemplo 4

2.5 Observações 5

3 A árvore de segmentos **7**

3.1 Construção 7

3.2 Atualização 7

3.3 Pergunta 8

4 Implementação da árvore de segmentos **9**

4.1 Ilustração da Implementação 9

4.2 Representação da Árvore 10

4.3 Construção 11

4.4 Atualização 13

4.5 Pergunta 14

4.6 União de dois nós 15

4.7 Variações de árvores de segmento ordinárias 15

4.7.1 Máximo 15

4.7.2 Máximo divisor comum 15

5 Propagação preguiçosa **17**

5.1 Representação da Árvore 17

5.2 Construção 18

5.3 Atualização 18

5.4 Pergunta 18

5.5 União de dois nós 19

5.6 Propagação 19

5.7 Aplicação da Propagação 19

6 Problemas Selecionados **21**

6.1 RPLN - Negative Score 21
6.1.1 URL 21
6.1.2 Descrição da entrada 21
6.1.3 Descrição da saída 21
6.1.4 Exemplo 22
6.1.5 Solução 22
6.2 INVCNT - Inversion Count 22
6.2.1 URL 22
6.2.2 Descrição da entrada 22
6.2.3 Descrição da saída 22
6.2.4 Exemplo 23
6.2.5 Solução 23
6.3 Potentiometers 23
6.3.1 URL 23
6.3.2 Descrição da entrada 24
6.3.3 Descrição da saída 24
6.3.4 Exemplo 24
6.3.5 Solução 25
6.4 Banco do Faraó 25
6.4.1 URL 25
6.4.2 Descrição da entrada 25
6.4.3 Descrição da saída 25
6.4.4 Exemplo 25
6.4.5 Solução 26
6.5 Interval Product 27
6.5.1 URL 27
6.5.2 Descrição da entrada 27
6.5.3 Descrição da saída 28
6.5.4 Exemplo 28
6.5.5 Solução 28
6.6 GSS3 - Can you answer these queries III 30
6.6.1 URL 30
6.6.2 Descrição da entrada 30
6.6.3 Descrição da saída 30
6.6.4 Exemplo 30
6.6.5 Solução 30
6.7 HORRIBLE - Horrible Queries 31

6.7.1 URL 31
6.7.2 Descrição da entrada 31
6.7.3 Descrição da saída 31
6.7.4 Exemplo 31
6.7.5 Solução 31

7 Conclusão **33**

Referências Bibliográficas **35**

Capítulo 1

Introdução

Como já pudemos adiantar no resumo, o presente trabalho de pesquisa científica versará sobre a estrutura de dados conhecida como Árvore de Segmentos.

Tal estrutura é comumente utilizada em programação competitiva por tratar-se de uma ferramenta versátil e extremamente importante na solução de determinados tipos de problema, sendo os mais comuns aqueles em que se deseja perguntar qual o elemento mínimo de um intervalo inúmeras vezes dado uma determinada sequência de dados.

No entanto, a pesquisa a que se propõe vai além de apenas demonstrar conceitos acerca da ferramenta mencionada acima.

Tendo em vista a escassez de material em língua portuguesa e a grande quantidade de material que demonstra apenas uma implementação "caixa preta", esta pesquisa pretende solidificar as formas e as possibilidades de utilização da ferramenta, bem como a problemática envolvida em sua escolha, as formas de implementação e, ainda, corroborar sua utilização através da solução de problemas práticos e ordinariamente enfrentados na esfera da programação competitiva.

Além disso, este estudo aprofunda-se, inclusive, em uma das formas de adaptação de implementação da árvores de segmentos, a propagação "preguiçosa", demonstrando o aumento de possibilidades de aplicação da ferramenta.

Capítulo 2

A escolha do problema

Na presente pesquisa demostraremos que o tipo de problema que gostaríamos de resolver é: dado um conjunto armazenado em uma estrutura de acesso aleatório aos seus índices (como um vetor) e uma operação associativa, desejamos responder eficientemente o resultado desta operação aplicada aos elementos de um intervalo desse conjunto múltiplas vezes.

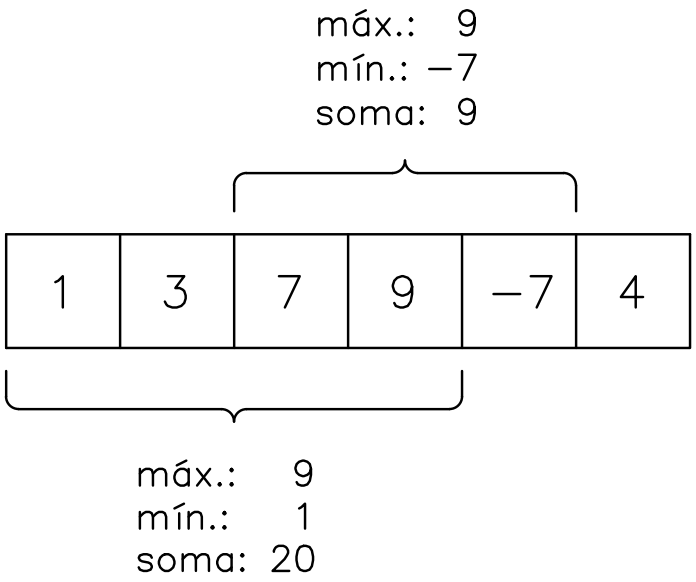

Figura 2.1: *Exemplo de estrutura, com operações aplicadas em intervalos.*

2.1 Motivação

A motivação deste trabalho revela-se no fato de que problemas que envolvem realizar múltiplas consultas sobre o resultado de uma operação associativa em intervalos de um dado conjunto com relação de ordem nos índices, são encontrados em programação competitiva. Tendo em vista a robustez dos métodos utilizados para resolvê-los, esse tipo de problema aparece com uma frequência um pouco menor que outros problemas que envolvem algoritmos mais simples, razão pela qual dominar a solução deles trará o diferencial nas competições de programação.

2.2 Objetivo

O objetivo deste trabalho é tornar o estudo deste tópico menos penoso, uma vez que existe muito pouco material em língua portuguesa e que a maioria se limita a apontar uma implementação "caixa preta"pouco detalhada. Além disso, este material pretende servir como referência para todos que têm interesse em maratona de programação ou que gostariam de estudar e se aprofundar no tópico.

2.3 Aplicação

Seguindo com as possibilidades de aplicação do método, faz-se necessário aprofundarmos um pouco mais nos tipos de problemas tratados. Assim, considerando que dados n itens de supermercado; cada item possui um preço e um peso. Além disso, os n itens estão organizados por categoria, ou seja, primeiro os laticínios, depois os de limpeza, etc. Como administradores do supermercado, gostaríamos de todos os dias saber quais são os produtos mais baratos de cada categoria para que possamos preencher cartazes de promoção, levando em conta o fato de que todos os dias os preços de alguns dos produtos podem mudar.

Como clientes, temos acesso a listas de produtos como esta de vários supermercados e, com isso, gostaríamos de saber, naquele dia, em qual supermercado a soma de produtos de uma determinada categoria é menor, levando em conta promoções relâmpago.

Na ciência da computação a estrutura de dados que nos ajuda a resolver este tipo de problema é chamada de Árvore de Segmentos; tal estrutura é extremamente versátil, baseada no paradigma de divisão e conquista, sendo possível idealizá-la como uma árvore usada para guardar informações sobre intervalos (segmentos) de um dado vetor (conjunto com relação de ordem nos índices) construída de tal forma a permitir perguntas e modificações a intervalos deste vetor de forma eficiente.

Uma das aplicações mais comuns desta estrutura envolve solucionar o problema do menor valor de um dado segmento, exemplificado acima pela lista de supermercado. Neste problema nos é dado um vetor com número e nos é perguntado inúmeras vezes qual é o menor valor de um dado intervalo.

2.4 Exemplo

Dado o seguinte vetor:

$$(1, 8, 2, 5, 4, 11, 3, 7)$$

Queremos saber o menor valor entre o terceiro e o quinto elemento: $min(2, 5, 4) = 2$

Depois gostaríamos de atualizar o valor do sexto elemento para 9, pois entramos em um período de promoção relâmpago para este produto, o que transforma o dado vetor em:

$$(1, 8, 2, 5, 4, 9, 3, 7)$$

E agora perguntar qual o elemento mais barato entre o sexto e o oitavo: $min(9, 3, 7) = 3$

Divisão e conquista

A ideia da solução utilizando o paradigma de divisão e conquista conserva-se no seguinte raciocínio:

- Se o intervalo que estamos analisando possui apenas um elemento então este elemento é trivialmente o menor do intervalo;

- Caso contrário, dividimos o intervalo em dois subintervalos de aproximadamente mesmo tamanho, tomamos o mínimo de cada um deles e, como a operação de mínimo é associativa, o mínimo deste intervalo é o mínimo entre os dois mínimos sub intervalos;

Ou seja:

$$a_i \text{ se } x = y$$
$$\text{ou}$$
$$min(f(x, \lfloor \tfrac{x+y}{2} \rfloor)), f(\lfloor \tfrac{x+y}{2} \rfloor + 1, y) \text{ caso contrario.}$$

Aplicações desta estrutura não se limitam a vetores de números inteiros e, frequentemente, são utilizados para resolver problemas em áreas como geometria computacional onde um ponto pode guardar muitas informações além de sua posição no plano.

2.5 Observações

Por outro lado, se tomarmos o já citado exemplo do supermercado e adicionarmos aos dados uma sacola com uma certa capacidade e perguntássemos qual é o subintervalo que cabe na sacola a considerar que a soma do peso de seus itens é menor ou igual a capacidade da sacola com a intensão de saber se isso maximiza a soma dos preços, infelizmente não seria possível encontrar a resposta através da utilização da árvore de segmentos, uma vez que a estrutura que aqui delineamos não resolve este problema diretamente.

Uma modificação simples que torna o problema bastante mais complexo, seria considerar um dado vetor de tamanho n e responder inúmeras vezes perguntas da seguinte natureza:

dado um intervalo e um número k, quantos elementos neste intervalo são maiores que k.

Imperioso salientar, ainda, que esta estrutura pode ser generalizada para mais do que uma dimensão.

Neste sentido, veremos a seguir as formas de implementação da árvore de segmentos, entendendo melhor sua versatilidade e formas de aplicação.

Capítulo 3

A árvore de segmentos

Apresentada a problemática em que se é possível utilizar a estrutura em análise, passamos a especificar suas características.

A árvore de segmentos é um estrutura de dados baseada no paradigma de divisão e conquista e pode ser pensada como uma árvore de intervalos avaliados em um vetor fundamental construído de forma a possibilitar perguntas sobre intervalos do vetor e modificações sobre estes mesmos intervalos de forma extremamente eficientes.

A estrutura da árvore de segmentos se baseia nos seguintes fatos:

- é uma árvore binária, ou seja, cada nó possui dois filhos;
- cada nó representa um intervalo do vetor fundamental e este nó pode conter informações sobre mais do que uma função aplicada em intervalos do vetor fundamental;
- cada folha da árvore está associada a um intervalo que contém apenas um elemento;
- subindo na árvore, cada pai representa a união dos resultados das funções aplicadas nos intervalos (disjuntos) representados por seus dois filhos;
- o nó raiz representa o vetor inteiro.

Para criar tal estrutura precisamos de três operações básicas, quais sejam:

3.1 Construção

Para construir uma árvore de segmentos precisamos inicializar valores nos nós desta, valores estes que representam o vetor fundamental. É possível criar a árvore de segmentos tanto de forma top-down (recursiva) ou bottom-up (iterativa). A construção top-down popula o valor do nó raiz que, por sua vez, implica em chamadas recursivas para cada uma das metades do intervalo que contém o intervalo em questão, resultando em outras chamadas para as respectivas metades; os casos base são as folhas, que podem ser calculados imediatamente dos valores originais do vetor. Uma vez calculados os valores das duas metades do intervalo, calcular o valor do pai se resume apenas em aplicar as funções nos respectivos resultados dos subintervalos.

3.2 Atualização

Para atualizar uma árvore de segmentos precisamos modificar o valor de algum elemento do vetor fundamental. Para isso modificamos o valor da folha referente a este elemento. As

outras folhas não são afetadas por essa atualização uma vez que cada folha está associada a apenas um elemento do vetor. Uma vez modificada a folha em questão o nó pai desta folha é afetado e pode ter seu valor modificado, assim como o nó avô e assim por diante até a raiz, mas nenhum outro nó é afetado. Novamente para executar uma atualização podemos optar pelos métodos top-down (recursivo) ou bottom-up (iterativo). Começamos fazendo a chamada na raiz, o que gera chamadas recursivas para apenas um dos filhos, aquele cujo intervalo contém o elemento que desejamos atualizar, o outro nó não é afetado. O caso base novamente é a folha associada ao elemento do vetor que desejamos atualizar. Depois de finalizada a recursão basta avaliar novamente o resultado da função dos filhos depois da atualização de um deles.

3.3 Pergunta

Para perguntar à uma árvore de segmentos, precisamos determinar o resultado de uma função aplicada a um determinado intervalo do vetor fundamental. A execução da pergunta é bastante complexa e acompanhará o exemplo. Assim, retomamos o exemplo do capítulo anterior:

Dado o vetor:

$$(1, 8, 2, 5, 4, 11, 3, 7)$$

queremos perguntar o menor valor entre o segundo e o quinto, incluso.

Representaremos essa pergunta da seguinte forma: $f(2, 5)$. Cada nó da árvore de segmentos contém o mínimo de algum intervalo, a raiz contém $f(1, 8)$ que é o menor valor do vetor inteiro, seu filho esquerdo possui $f(1, 4)$ e seu filho direito $f(5, 8)$. Podemos perceber que $f(2, 5) = min(f(2, 2), f(3, 4), f(5, 5))$.

Conforme explanado anteriormente, são diversas as formas de utilização da árvore de segmentos, sendo natural que, no decorrer das etapas e treinamentos de programação competitiva, sua utilização se torne intuitiva para os casos aplicáveis.

A seguir, veremos como aplicá-la de forma prática.

Capítulo 4

Implementação da árvore de segmentos

Neste tópico iremos expor implementações da árvore de segmentos vistas no capítulo anterior. As implementações estão na linguagem C++ e os códigos podem ser obtidos em https://linux.ime.usp.br/matheusmso/mac0499/codigos.html.

Embora o foco das implementações seja a resolução de problemas no estilo programação competitiva, preferimos dar mais atenção à facilidade de entendimento em detrimento à eficiência e redução de linhas de código. Por essa razão, aconselhamos o leitor, visando montar sua biblioteca pessoal de algoritmos, que reescreva as implementações de acordo com suas necessidades e seus estilos.

4.1 Ilustração da Implementação

Visando a ilustração da implementação da árvore de segmentos, partiremos do pressuposto contido no seguinte problema:

Em uma cidade distante e com 100 mil habitantes, o Secretário de Educação, preocupado com alguns índices requereu que fosse feito um levantamento sobre o indivíduo mais novo entre diversos intervalos da sociedade. Deveremos considerar ainda que haverá alterações na lista de idade dos habitantes do município considerando-se a saída de velhos moradores e a chegada de novos integrantes.

Assim, tendo a lista de idade de todos os habitantes do município, nosso programa precisa retornar a informação do indivíduo mais novo, quando perguntado inúmeras vezes sobre diversos intervalos.

Caso o número de perguntas e atualizações seja pequeno (100 perguntas ou atualizações), poderíamos simplesmente, para cada pergunta ou atualização, caminhar por toda a lista para obter o resultado, sem que fosse necessário nenhum pré processamento.

Esse conceito de número pequeno de perguntas e atualizações - 100 perguntas ou atualizações, deriva do fato de que um processador moderno (presente em todos os juízes de programação competitiva) executa em torno de 10 milhões (100 x 100.000) de operações por segundo.

Tratando-se da necessidades de perguntas ou atualizações superiores a 100, a presente estrutura de dados se torna necessária, haja vista que utilizando-a o tempo necessário para cada pergunta ou atualização seja reduzido de $O(N)$ para $O(\log N)$.

Tendo em mãos os dados necessários e sabendo da possibilidade da utilização da árvore de Segmentos, passaremos a representá-la na sequencia.

4.2 Representação da Árvore

Usaremos um vetor para representar a árvore de segmentos, nesse caso, sabemos que o conjunto original que nos interessa perguntar mínimos frequentemente tem 100 mil elementos.

```
1 const int N = (int)1e5+7;
2 const int INF = 0x3f3f3f3f;
3 const int NEUTRAL = INF;
4 int segtree[4*N];
```

Acima, declaramos uma variável INF, disposta desta maneira por diversas razões que passamos a expor:

- Foma hexadecimal; a escrita na base 10 é demasiada propensa a erros;
- Trata-se de um número bastante grande (na mesma ordem de grandeza que $0x3f3f3f3f_{(16)} = 1061109567_{(10)}$ $0x7fffffff_{(16)} = 2147483647_{(10)}$);
- Embora seja um número muito extenso, é possível realizar operações em si próprio sem causar overflow (INF + INF = não causa overflow);
- É uma repetição de bytes iguais, o que permite a utilização da função memset para atribuir valores casas de vetores à ele.

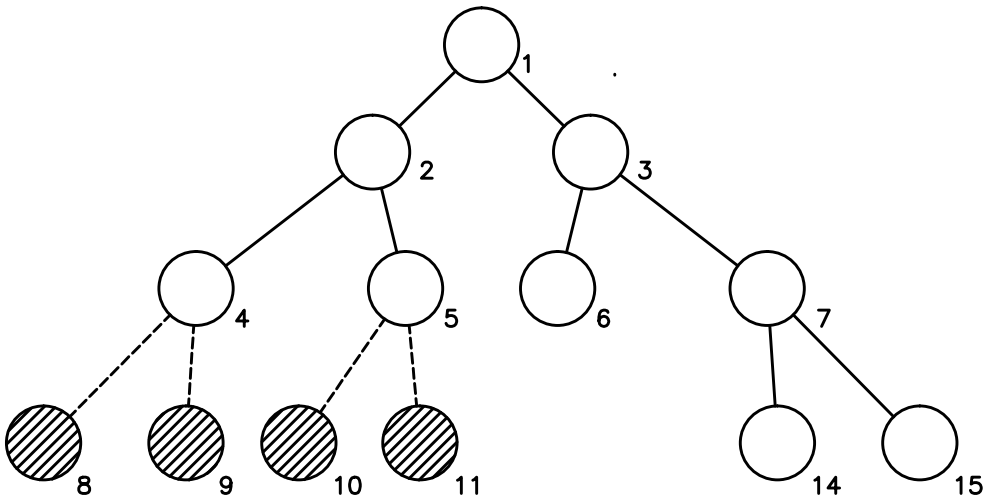

Figura 4.1: *Pior caso para $n = 5$.*

Por se tratar de uma operação onde se deseja encontrar o mínimo, é necessário que haja um número neutro na operação e, neste caso, trata-se de um número fora do universo aqui discutido, ou seja, infinito. Exemplo: Tendo uma idade qualquer, 42, $min(42, INF) = min(INF, 42) = 42$ e isso é verdade para qualquer número dentro deste universo (idades - 0 - 150 anos).

Aqui, declaramos um vetor de tamanho igual a quatro vezes o número de elementos do conjunto original, isso se dá pelo seguinte:

A lista original mapeia diretamente para as folhas da árvore, portanto o número de folhas é pelo menos o número de elementos da lista de dados. Uma lista com 2^x elementos, gera uma árvore com pelo menos 2^x folhas, ou seja, uma árvore com $2^{(x+1)} - 1$ elementos, um vez que a árvore de segmentos é uma árvore binária perfeita. Porém, poderemos ter folhas inúteis no caso em que o tamanho da lista não for uma potência de 2. No pior caso será uma lista de tamanho $2^x + 1$, onde teremos $2N$ folhas pois temos que arredondar para a próxima potência de 2. Dessa forma uma árvore binária perfeita com $2N$ folhas terá aproximadamente $4N$ nós no total.

Um exemplo do pior caso está na figura 4.1.

Como visto no capítulo anterior, vamos implementar as três principais funções dessa estrutura: construção, atualização e pergunta de forma recursiva. Por fim, trataremos da união dos nós.

4.3 Construção

Tendo em vista todo o explanado acima, segue a construção da árvore de segmentos, haja vista que desejamos popular o vetor.

```
1 void build(int node = 1, int l = 0, int r = n) {
2     if (l + 1 == r) {
3         segtree[node] = v[l];
4         return;
5     }
6     int mid = (l + r)/2;
7     build(2*node, l, mid);
8     build(2*node+1, mid, r);
9     segtree[node] = join(segtree[2*node], segtree[2*node+1]);
10 }
```

Esta função deve ser chamado logo após a leitura da lista de idades, aqui representada por v. Para simular, vamos supor a seguinte lista apresentada:

$$(5, 3, 11, 7, 1)$$

A ideia do algorítimo é iniciar pelo nó que representará a árvore em sua totalidade e chamar recursivamente para as metades de, aproximadamente, mesmo tamanho, sendo o caso base quando só há um sub elemento na sub árvore para qual a função está sendo chamada.

É importante notar que para esta implementação estamos utilizando uma convenção de intervalos fechados à esquerda e abertos à direita ([)).

Simularemos, então a construção da árvore para a lista apresentada acima:

A primeira chamada $l = 0$ e $r = n$. Este intervalo tem tamanho 5 e portanto calcularemos a sua metade, $mid = 2$ o que resulta em duas chamadas, uma para o nó 2, com $l = 0$ e $r = 2$ e outra para o nó 3 com $l = 2$ e $r = 5$.

A chamada para o nó 2 se desdobrará em duas chamadas para os nós 4 e 5, ambos com intervalos de tamanho 1 que, portanto, retornarão seus elementos na lista de idades, 5 e 3,

o que resulta neste sendo valorado em 3 que é o $min(5, 3)$.

Até o momento temos a seguinte situação:

Lista de idades:

$$(5, 3, 11, 7, 1)$$

Árvore de segmento:

$$(-, -, 3, -, 5, 3)$$

Vamos continuar com a chamada para o nó 3. Esta por sua vez será subdividida em duas chamadas, para o nó 6 com $l = 2$ e $r = 3$ e para o nó 7 com $l = 3$ e $r = 5$. O nó 6 retornará a casa 2 da lista de idades, por ter tamanho unitário. Já a chamada para o nó 7 terá de ser subdividida uma vez mais para os elementos 3 e 4 da lista, 7 e 1 (nós 14 e 15). Com isso o nó 7 resolve para 1 e o nó 3 para 1 também pois $1 = min(1, 11)$.

Lista de idades:

$$(5, 3, 11, 7, 1)$$

Árvore de segmento:

$$(-, 1, 3, 1, 5, 3, 11, 1, -, -, -, -, -, -, 7, 1)$$

Árvore criada para essa lista esta na figura 4.2.

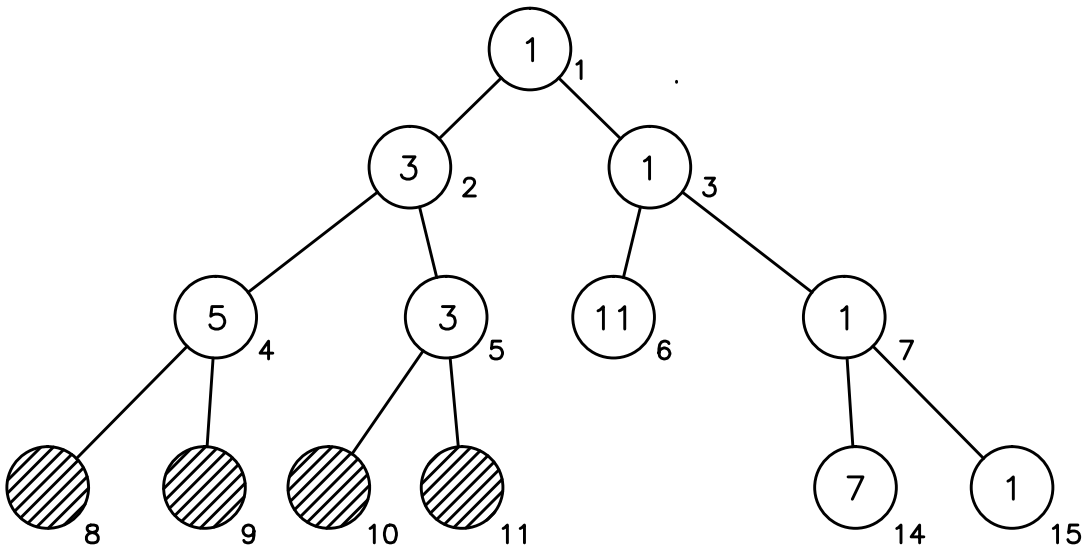

Figura 4.2: *Representação dessa árvore.*

Com isso a árvore de segmentos está construída. Importante notar que a casa 0 da árvore permanece sem uso para o essa implementação da estrutura.

Como verificado, cada elemento da lista de idades e cada casa do vetor que representa a árvore de segmentos é acessado somente uma vez portanto a construção da árvore tem gasto de tempo de ordem linear no tamanho da lista de idades $O(N)$.

4.4 Atualização

Explanado o campo da construção da estrutura da árvore de segmentos, passamos a apreciar sua atualização.

```
1 void update(int pos, int value, int node = 1, int l = 0, int r = n) {
2     if (l + 1 == r) {
3         v[pos] = value;
4         return;
5     }
6     int mid = (l + r)/2;
7     if (pos < mid)
8         update(pos, value, 2*node, l, mid);
9     else update(pos, value, 2*node+1, mid, r);
10    segtree[node] = join(segtree[2*node], segtree[2*node+1]);
11 }
```

Sempre que houver necessidade em atualizar alguma das idades da lista de idades, essa função deverá ser chamada.

Suponhamos que gostaríamos de atualizar a 4ª casa da lista em questão, de 1 para 17:

Lista original

$$(5, 3, 11, 7, 1)$$

Lista atualizada

$$(5, 3, 11, 7, 17)$$

Esta atualização seria encapsulada da seguinte forma:$update(4, 17)$.

Vamos, então, simular esta atualização. A chamada é inicializada com $l = 0$ e $r = 5$ e com isso $mid = 2$.

Como gostaríamos de atualizar a casa 4 seguiremos com a chamada para o nó 3, $(4, 17, 3, 2, 5)$, ou seja, não olharemos para o nó 2, nem para nenhum nó contido em sua sub árvore.

Uma vez no nó 3, $mid = 3$, logo seguiremos para o nó 7, pois $4 \geq 3$.

Por ordem, iremos para o nó 15, pois $4 \geq 4$. Este nó representa um intervalo unitário e, exatamente, a folha que gostaríamos de atualizar que resultará na atualização de todos os nós decorrentes deste, pelo caminho, até a raiz, que, neste caso, serão os nós $15, 7, 3, 1$, fazendo com que a árvore fique da seguinte forma:

Lista de idades:

$$(5, 3, 11, 7, 17)$$

Árvore de segmento:

$$(-, 3, 3, 7, 5, 3, 11, 7, -, -, -, -, -, -, 7, 17)$$

Árvore depois da atualização esta na figura **??**.

Como podemos perceber essa função gasta tempo proporcional aos nós que precisam ser inspecionados durante a atualização, que são sempre algum caminho de alguma folha até a raiz. Como a árvore de segmento é uma árvore binária, sua altura é da ordem de $O(\log N)$.

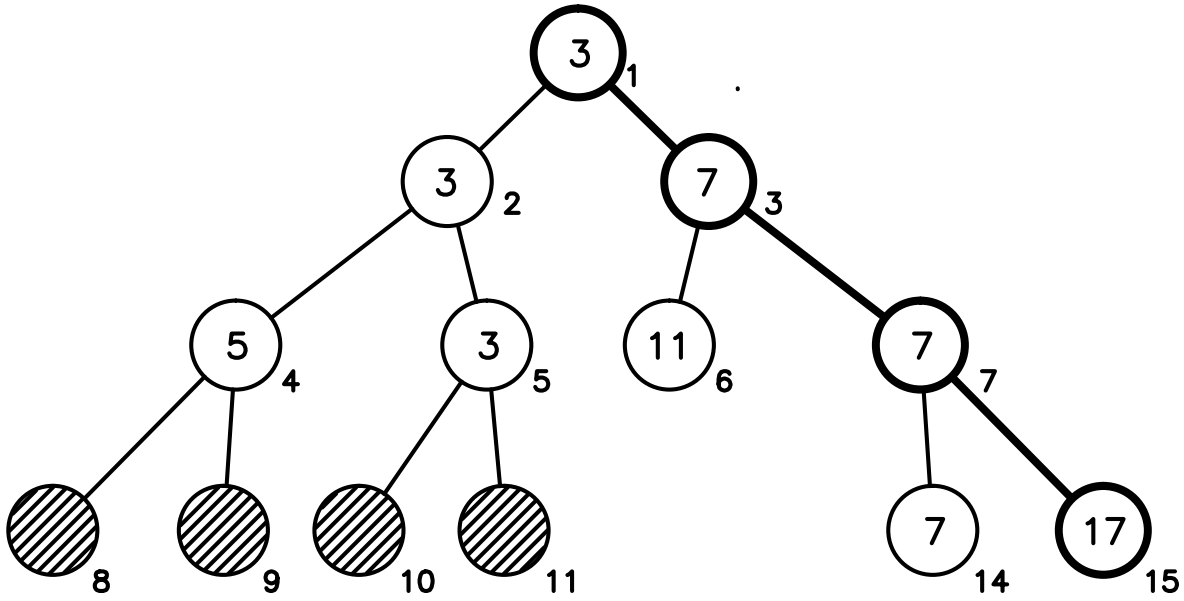

Figura 4.3: *Árvore atualizada.*

4.5 Pergunta

Conforme vimos desde o início do capítulo, nosso objetivo é encontrar o valor mínimo dado um determinado intervalo (ou vários) utilizando uma estrutura de dados denominada de árvore de segmentos. Para tanto, já delineamos a definição dos dados a serem analisados apresentando aspectos da construção da árvore de segmentos e sua atualização. Partiremos, agora, para as perguntas possíveis dados os intervalos e a lista de dados apresentada.

```
1 int query(int x, int y, int node = 1, int l = 0, int r = n) {
2     if (x >= r || y <= l) return NEUTRAL;
3     if (x <= l && y >= r) return segtree[node];
4     int mid = (l + r)/2;
5     return join(query(x, y, 2*node, l, mid), query(x, y, 2*node+1, mid, r)
          );
6 }
```

Suponha que desejamos saber o índice do habitante mais novo entre os habitantes de índices 2 e 4 da lista original. Para isso chamamos $query(2, 5)$. Lista de idades:

$$(5, 3, 11, 7, 1)$$

Por inspeção sabemos que a resposta é o habitante da casa número 4, de idade 1.

Vamos simular o algoritmo que determina essa resposta, ou seja, partiremos da chamada $query(2, 5)$. Como o intervalo de interesse $[2, 5)$ não está totalmente disjunto, primeiro

caso base, nem contém totalmente o intervalo que estamos observando, segundo caso base, calcularemos a resposta para os dois sub intervalos, $[0, 2)$ e $[2, 5)$.

O intervalo de interesse continua sendo $[2, 5)$, e nesse caso a primeira chamada recursiva está totalmente disjunta e portanto retorna o valor neutro. Já a segunda chamada está totalmente contido e portanto retorna o valor do nó 3, que representa exatamente esse intervalo, $[2, 5)$. Com isso o retorno de $query(2, 5)$ é $min(INF, 1) = 1$.

A operação supra mencionada também passa por um caminho entre uma folha e a raiz e portanto tem gasto de tempo proporcional a altura da árvore $O(\log N)$.

4.6 União de dois nós

Essa operação pode parecer redundante nesse caso, porém uma vez que um nó guardar mais informações faz sentido existir uma função para resolver a união de dois nós. Dessa forma podemos obter com a mesma implementação uma árvore de segmentos de mínimo, máximo ou máximo divisor comum entre outros mudando somente a função de união e a constante neutra da sessão anterior.

```
1 int join(int a, int b) {
2     return min(a, b);
3 }
```

De acordo com o objetivo, no caso em questão, gostaríamos apenas de saber o menor valor do intervalo, logo, a operação de união de dois nós se resume a devolver o mínimo entre eles.

4.7 Variações de árvores de segmento ordinárias

De acordo com tudo que foi discutido até este ponto da pesquisa, já foi possível compreender que é possível implementar variações, que façam com que a estrutura suporte operações diferentes. É o que veremos a seguir.

4.7.1 Máximo

```
1 const int NEUTRAL = -INF;
2 int join(int a, int b) {
3     return max(a, b);
4 }
```

Nesse caso queremos retornar o maior elemento do intervalo, e para isso basta que a operação de união dos nós se resuma a retornar o máximo entre os elementos e que o elemento neutro seja o infinito negativo, uma vez que $max(-INF, 42) = max(42, -INF) = 42$.

4.7.2 Máximo divisor comum

```
1 const int NEUTRAL = 0;
2 int join(int a, int b) {
3     return __gcd(a, b);
4 }
```

Nesse caso queremos retornar o máximo divisor comum dos elementos do intervalo, e para isso basta que a operação de união dos nós se resuma a retornar o máximo divisor comum entre os elementos e que o elemento neutro seja o 0, uma vez que $gcd(0, 42) = max(42, 0) = 42$.

Capítulo 5

Propagação preguiçosa

Uma das formas de adaptação da árvore de segmentos, de forma a otimizar a utilização da ferramenta e maximizar sua aplicação é a Lazy Propagation ou propagação preguiçosa.

Como o próprio nome sugere, a propagação preguiçosa é a forma de propagar a atualização de um determinado intervalo na árvore apenas quando for necessária sua utilização sem que seja necessário caminhar por todos os nós da árvore fazendo com que o gasto de tempo do algorítimo seja menor.

A base da propagação lenta se dá através da marcação do nó na árvore que deve ser alterado, uma vez que ambas as árvores desfrutam do mesmo índice e, quando existe a necessidade de consultar os dados de um segmento, verifica-se na árvore auxiliar se há marcação em algum nó; havendo a marcação, faz-se atualização dos nós filhos até a profundidade que descer para a consulta, zerando-se o nó da árvore auxiliar e marcando a atualização apenas no nó que esteve pendente de atualização.

Para exemplificar de maneira didática o funcionamento de propagação preguiçosa, tomaremos como exemplo a possibilidade de termos diversos baldes com uma determinada quantidade (inteira) de litros de água: Lista de baldes:

$$(3, 2, 5, 7, 9)$$

Agora, desejamos adicionar 2 litros apenas nos baldes de que pertençam ao intervalo $[1, 2]$, tendo então, a seguinte lista atualizada:

$$(3, 4, 7, 7, 9)$$

Passaremos a expor, então, a construção da árvore de segmentos com a implementação da propagação preguiçosa.

5.1 Representação da Árvore

Para representar o universo desta árvore de segmentos, nesse caso, sabemos que o conjunto original que nos interessa partirá de no máximo 100 mil elementos (assim como na implementação original), dentre os quais realizaremos as perguntas de soma dos elementos de um dado intervalo, tendo em vista que estamos tratando de uma forma de adaptação da árvore de segmentos.

```
1 const int N = (int)1e5+7;
2 const int NEUTRAL = 0;
3 int segtree[4*N];
4 int lazy[4*N]
```

Além do vetor da árvore que já existia para a representação da árvore de segmento, alocaremos um segundo vetor para guardar a informação a ser propagada pelos nós de forma preguiçosa. Configuramos a variável neutra para 0 pois estamos lidando com somas, nesse caso.

5.2 Construção

Mantêm-se o mesmo raciocínio da construção da árvore de segmentos visto no item 4.3, uma vez que a estrutura de construção da árvore não precisa ser modificada para esta adaptação.

É necessário apenas um segundo vetor como citado acima.

5.3 Atualização

A atualização sofre alterações sutis.

```
1 void update(int x, int y, int value, int node = 1, int l = 0, int r = n) {
2     if (x >= r || y <= l) return;
3     if (x <= l && y >= r) {
4         app(node, l, r, value);
5         return;
6     }
7     shift(node, l, r);
8     int mid = (l + r)/2;
9     update(x, y, value, 2*node, l, mid);
10    update(x, y, value, 2*node+1, mid, r);
11    segtree[node] = join(segtree[2*node], segtree[2*node+1]);
12 }
```

Uma vez que estamos trabalhando com atualização em intervalo e não de um elemento como no caso anterior, precisamos de uma lógica muito similar à utilizada na seção 4.5 do capítulo anterior.

Uma vez aplicada a lógica de intervalos, as alterações se restringem às linhas 4 e 7. A nova função chamada na linha 4, *app* tem o objetivo de aplicar as mudanças para os filhos deste nó caso o intervalo de interesse contenha completamente o intervalo representado por ele. A função $shift$, chamada na linha 7, tem como responsabilidade propagar as mudanças para os filhos deste nó. Como veremos a chamada da função de propagação é necessária tanto na atualização como na pergunta, pois, uma vez que é necessário caminhar por um dado nó, executar uma ação a mais não altera assintoticamente o desempenho da mesma.

5.4 Pergunta

A atualização é basicamente a mesma da árvore original; precisamos apenas introduzir a chamada da função de propagação conforme citado na seção anterior.

```
1 int query(int x, int y, int node = 1, int l = 0, int r = n) {
2     if (x >= r || y <= l) return NEUTRAL;
3     if (x <= l && y >= r) return segtree[node];
4     shift(node, l, r);
5     int mid = (l + r)/2;
6     return join(query(x, y, 2*node, l, mid), query(x, y, 2*node+1, mid, r)
          );
```

```
7 }
```

Como veremos, a função de propagação tem ação atômica, $O(1)$ e portanto, sua chamada dentro das funções de atualização e pergunta não altera a complexidade original de tais funções e portanto ambas continuam tendo gasto de tempo proporcional a $O(\log N)$.

5.5 União de dois nós

Se resume a uma soma nesse caso.

```
1 int join(int a, int b) {
2     return a + b;
3 }
```

5.6 Propagação

Caso a propagação seja necessária, a responsabilidade desta função será propagar a informação do presente nó para seus filhos.

```
1 void shift(int node, int l, int r) {
2     int mid = (l + r)/2;
3     if (lazy[node] != 0) {
4         app(2*node, l, mid, lazy[node]);
5         app(2*node+1, mid, r, lazy[node]);
6     }
7     lazy[node] = 0;
8 }
```

Essa função propaga a informação para seus filhos chamando a função de aplicação dessa propagação nestes. Uma vez aplicada devemos marcar esse nó como propagado, isso é feito na linha 7.

5.7 Aplicação da Propagação

```
1 void app(int node, int l, int r, int x){
2   lazy[node] += x;
3   segtree[node] += (r - l) * x;
4 }
```

Finalmente aplicaremos a propagação para um dado nó; essa operação é simplesmente somar neste o valor que desejamos adicionar no intervalo, vezes o tamanho do intervalo representado por esse nó e marcar o aumento necessário no nó de propagação para que esse valor seja propagado para seus filhos quando necessário.

Capítulo 6

Problemas Selecionados

Neste item dissertaremos acerca de vários exemplos de problemas de programação competitiva cuja solução envolve a utilização da estrutura de dados árvore de segmentos. Nos primeiros problemas encontraremos a solução com a aplicação direta da implementação da árvore de segmentos, não sendo necessário comentários adicionais, haja vista a vasta explanação anterior. Através da solução destes problemas, será possível que os estudantes de programação competitiva verifiquem seu entendimento sobre o tema e se suas implementações dos algoritmos estão corretas. Os problemas que se seguirão versarão sobre a implementação da árvore de segmentos com a propagação preguiçosa. Todos os problemas estão listados em ordem de dificuldade e possuem um link para um juiz online com o respectivo problema, para que o estudante interessado possa submeter e verificar sua própria solução. Além disso, uma implementação em C++ da solução de cada problema está disponível em https://linux.ime.usp.br/~matheusmso/mac0499/codigos.html.

6.1 RPLN - Negative Score

Este exemplo demonstra a aplicação direta da estrutura estudada neste trabalho, mais precisamente a detalhada no capítulo 4. Importante notar que esse problema não requer implementação da funcionalidade de atualização.

6.1.1 URL

https://www.spoj.com/problems/RPLN/

6.1.2 Descrição da entrada

A primeira linha dos dados de teste começará com um inteiro T representando o número de casos, cada um dos casos começa com dois inteiros N e Q. N inteiros seguirão indicando a pontuação em cada avaliação, depois disso, Q consultas serão iniciadas, cada consulta consistirá em dois inteiros A e B, o intervalo de interesse.

6.1.3 Descrição da saída

Você deve produzir a linha que indica qual teste estamos imprimindo e, em seguida, o resultado de cada consulta, lembre-se, que estamos em busca da pior pontuação da avaliação A à avaliação B, inclusive.

6.1.4 Exemplo

Entrada:

```
2
5 3
1 2 3 4 5
1 5
1 3
2 4
5 3
1 -2 -4 3 -5
1 5
1 3
2 4
```

Saída:

```
Scenario #1:
1
1
2
Scenario #2:
-5
-4
-4
```

6.1.5 Solução

Basta utilizar a estrutura estudada, em sua forma minimal, para responder as perguntas de mínimo em um dado intervalo.

6.2 INVCNT - Inversion Count

6.2.1 URL

https://www.spoj.com/problems/INVCNT/

6.2.2 Descrição da entrada

A primeira linha contém T, o número de casos de teste seguido por um espaço em branco. Cada um dos testes começa com um número N. Cada uma das linhas que seguem contém um dos elementos do vetor que gostaríamos de contar o número de inversões.

6.2.3 Descrição da saída

Para cada saída de teste, uma linha dando o número de inversões de A.

6.2.4 Exemplo

Entrada:

```
2

3
3
1
2

5
2
3
8
6
1
```

Saída:

```
2
5
```

6.2.5 Solução

Neste problema gostaríamos de responder a seguinte pergunta: dado um número x quantos números maiores que x já estão presentes? Como os números estão limitados a 10^7 podemos alocar um vetor booleano que marcará se um número está ou não presente na lista. Usando essa estrutura, resolver o problema se resume a processar os números em ordem e perguntar para cada um quantos maiores que ele existem, somar isso a resposta e depois adicioná-lo à estrutura.

Para atingir esse resultado basta alterarmos um pouco a função de atualização:

```
1 void update(int pos, int value, int node = 1, int l = 0, int r = N) {
2   if (l + 1 == r) {
3     segtree[node]++;
4     return;
5   }
6   int mid = (l + r)/2;
7   if (pos < mid)
8     update(pos, value, 2*node, l, mid);
9   else
10    update(pos, value, 2*node+1, mid, r);
11  segtree[node] = join(segtree[2*node], segtree[2*node+1]);
12 }
```

Importante notar que para esse caso, dado que não existe uma lista de origem, não precisamos construir a árvore, ou seja, não precisamos implementar a função *build*.

6.3 Potentiometers

6.3.1 URL

https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&page=

show_problem&problem=192

6.3.2 Descrição da entrada

Cada caso começa com N, o número de potenciômetros no array. Cada uma das próximas N linhas contém um número entre 0 e 1000, as resistências iniciais dos aparelhos na ordem 1 a N. Cada uma das linhas contém uma ação. Existem três tipos de ação:

- "S x r- define o aparelho x para r Ohms. x é um número válido de potenciômetro e r é entre 0 e 1000.
- "M x y- meça a resistência entre o terminal esquerdo do potenciômetro x e o terminal direito de potenciômetro y. Ambos os números serão válidos e x é menor que ou igual a y.
- "END- final deste caso. Aparece apenas uma vez no final de uma lista de ações.

$N = 0$ indica o fim dos casos de teste.

6.3.3 Descrição da saída

Para cada caso na entrada, produza uma linha que indique o caso de teste em questão. Para cada medição na entrada, imprima uma linha contendo um número: a resistência medida em Ohms. As ações devem ser aplicadas ao array de potenciômetro na ordem dada na entrada.

6.3.4 Exemplo

Entrada:

```
3
100
100
100
M 1 1
M 1 3
S 2 200
M 1 2
S 3 0
M 2 3
END
10
1
2
3
4
5
6
7
8
9
10
```

M 1 10
END
0

Saída:
Case 1:
100
300
300
200
Case 2:
55

6.3.5 Solução

Esse problema também é bastante direto, precisamos atualizar elementos da lista e responder para perguntas do tipo qual a soma deste sub intervalo.

Para isso basta utilizar a estrutura detalhada no capítulo 4 com a função de união de dois nós apresentada na seção 5.5.

6.4 Banco do Faraó

6.4.1 URL

https://br.spoj.com/problems/BANFARAO/

6.4.2 Descrição da entrada

A entrada é composta por diversas instâncias. A primeira linha da entrada contém um inteiro T indicando o número de instâncias. A primeira linha de cada instância contém um inteiro N, indicando o número de contas no Banco do Faraó. A segunda linha de cada instância contém N inteiros, entre -10000 até 10000, indicando os saldos nas contas dos correntistas. A terceira linha contém um inteiro Q indicando o número de consultas que serão feitas. Cada uma das Q linhas seguintes contém dois inteiros A e B indicando o intervalo que deve ser consultado.

6.4.3 Descrição da saída

Para cada instância seu programa deve produzir Q linhas na saída, sendo uma para cada consulta. Cada uma dessas linhas deve conter dois inteiros: o primeiro representa a soma do intervalo com maior soma, e o segundo, o número de elementos desse intervalo. Caso haja mais de um intervalo com maior soma, imprima o número de elementos naquele com maior número de elementos.

6.4.4 Exemplo

Entrada:
3

```
3
-1 -2 -3
1
1 1
8
1 2 -1 4 9 8 -1 2
4
1 3
1 4
2 5
7 8
3
0 0 0
1
1 3
```

Saída:

```
-1 1
3 2
6 4
14 4
2 1
0 3
```

6.4.5 Solução

Para responder qual o subintervalo de maior soma e quantos elementos ele possúi, precisamos guardar para cada intervalo do vetor as seguintes informações:

- a soma do subintervalo com maior soma do intervalo e seu tamanho.
- a maior soma de prefixo do intervalo e seu tamanho.
- a maior soma de sufixo do intervalo e seu tamanho.
- a soma total do intervalo e seu tamanho.

Para isso definimos o nó da árvore da seguinte forma:

```
1 struct node {
2   pair<int, int> bst;
3   pair<int, int> pre;
4   pair<int, int> suf;
5   pair<int, int> tot;
6 };
7 node seg[4*N];
```

Dessa forma podemos usar a estrutura definida no capítulo 4 e basta adaptar a função que une dois nós. Dados dois nós para uni-los basta fazer algumas comparações:

- a soma do subintervalo com maior soma desse novo intervalo é o máximo entre 3 fatores, o subintervalo de maior valor do nó esquerdo, o subintervalo de maior valor do nó direito e a soma do maior sufixo do nó esquerdo com o maior prefixo do nó direito.

- a maior soma de prefixo desse novo intervalo é o máximo entre a maior soma de prefixo do nó esquerdo e o total do nó esquerdo somado com a maior soma de prefixo do nó direito.
- a maior soma de sufixo desse novo intervalo é o máximo entre a maior soma de sufixo do nó direito e o total do nó direito somado com a maior soma de sufixo do nó esquerdo.
- a soma total desse novo intervalo é simplesmente a soma dos totais dos nós esquerdo e direito.

```
1 node join(node a, node b) {
2   if (a.bst.second == 0) return b;
3   if (b.bst.second == 0) return a;
4   node aux;
5   aux.tot = a.tot+b.tot;
6   aux.bst = max({a.bst, b.bst, a.suf+b.pre});
7   aux.pre = max(a.pre, a.tot+b.pre);
8   aux.suf = max(b.suf, b.tot+a.suf);
9   return aux;
10 }
```

As duas primeira linhas da função tratam os casos extremos onde um dos nós representa um intervalo vazio.

E a criação de um novo nó segue a seguinte prática uma vez que um elemento é o total, melhor prefixo sufixo e soma.

```
1 void build(int no = 1, int l = 0, int r = n) {
2   if (l + 1 == r) {
3     seg[no].bst = make_pair(v[l], 1);
4     seg[no].pre = make_pair(v[l], 1);
5     seg[no].suf = make_pair(v[l], 1);
6     seg[no].tot = make_pair(v[l], 1);
7     return;
8   }
9   int mid = (l + r)/2;
10  build(2*no, l, mid);
11  build(2*no+1, mid, r);
12  seg[no] = join(seg[2*no], seg[2*no+1]);
13 }
```

Esse problema não necessita a implementação da função de atualização. O problema GSS3 é bastatnte similar porém exige essa funcionalidade.

6.5 Interval Product

6.5.1 URL

https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=3977

6.5.2 Descrição da entrada

A primeira linha contém dois inteiros N e k, indicando respectivamente o número de elementos na sequência e o número de rodadas do jogo. A segunda linha contém N inteiros x_i que representam os valores iniciais da sequência. Cada uma das próximas k linhas descreve

um comando e começa com uma letra maiúscula que é "C"ou "P". Se a letra for 'C', a linha descreve um comando de mudança e a letra é seguida por dois inteiros i e v indicando que x_i deve receber o valor v. Se a letra for "P", a linha descreve um comando do produto e a letra é seguida por dois inteiros i e j indicando que o produto de x_i a x_j, inclusive, deve ser calculado. Dentro de cada caso de teste, há pelo menos um comando do produto.

6.5.3 Descrição da saída

Para cada caso de teste, imprima uma linha com uma cadeia representando o resultado de todos os comandos do produto no caso de teste. O i-ésimo caractere da string representa o resultado do i-ésimo comando do produto. Se o resultado do comando for positivo, o caractere deve ser "+"(mais); se o resultado for negativo, o caractere deve ser -"(menos); se o resultado for zero, o caractere deve ser "0"(zero).

6.5.4 Exemplo

Entrada:

4 6
-2 6 0 -1
C 1 10
P 1 4
C 3 7
P 2 2
C 4 -5
P 1 4
5 9
1 5 -2 4 3
P 1 2
P15
C 4 -5
P 1 5
P 4 5
C 3 0
P 1 5
C 4 -5
C 4 -5

Saída:

0+-
+-+-0

6.5.5 Solução

Para responder se o resultado do produto de um intervalo é positivo negativo ou zero basta que guardemos duas informações:

- quantos números negativos existem num dado intervalo.
- quantos zeros existem num dado intervalo.

Assim basta usarmos duas árvores se segmento, muito parecidas com a utilizada no problema 6.2, ou seja, uma árvore de segmento que guarda qutos elementos existem num dado intervalo, uma árvore de segmento de soma onde o vetor de origem é 1 se o elemento está presente e 0 caso contrário.

Para isso alteramos a função de construção para representar essa lógica:

```
1 void build(int node = 1, int l = 0, int r = n) {
2   if (l + 1 == r) {
3     if (v[l] == 0)
4       segz[node] = 1;
5     else if (v[l] < 0)
6       segn[node] = 1;
7     else
8       segz[node] = segn[node] = 0;
9     return;
10  }
11  int mid = (l + r)/2;
12  build(2*node, l, mid);
13  build(2*node+1, mid, r);
14  segn[node] = join(segn[2*node], segn[2*node+1]);
15  segz[node] = join(segz[2*node], segz[2*node+1]);
16 }
```

onde *segz* é a árvore que representa o vetor de presença de zeros e *segn* a árvore que representa a presença de negativos.

A atualização também precisa refletir essa mudança:

```
1 void update(int pos, int value, int node = 1, int l = 0, int r = n) {
2   if (value > 0) {
3     segn[node] -= v[pos] < 0;
4     segz[node] -= v[pos] == 0;
5   }
6   else if (value < 0) {
7     segn[node] += v[pos] >= 0;
8     segz[node] -= v[pos] == 0;
9   }
10  else {
11    segn[node] -= v[pos] < 0;
12    segz[node] += v[pos] != 0;
13  }
14  if (l + 1 == r) {
15    v[pos] = value;
16    return;
17  }
18  int mid = (l + r)/2;
19  if (pos < mid)
20    update(pos, value, 2*node, l, mid);
21  else
22    update(pos, value, 2*node+1, mid, r);
23  segz[node] = join(segz[2*node], segz[2*node+1]);
24  segn[node] = join(segn[2*node], segn[2*node+1]);
25 }
```

6.6 GSS3 - Can you answer these queries III

6.6.1 URL

https://www.spoj.com/problems/GSS3/

6.6.2 Descrição da entrada

A primeira linha de entrada contém um inteiro N. A linha a seguir contém N inteiros, representando a sequencia $a_1..a_N$. A terceira linha contém um inteiro M. As próximas M linhas contêm as operações no seguinte formato:

- 0 x y: modifica a_x para y.
- 1 x y: impressão $max[a_i + a_{i+1} + .. + a_j | x \leq i \leq j \leq y$.

6.6.3 Descrição da saída

Para cada consulta, imprima um inteiro como o problema requerido.

6.6.4 Exemplo

Entrada:

```
4
1 2 3 4
4
1 1 3
0 3 -3
1 2 4
1 3 3
```

Saída:

```
6
4
-3
```

6.6.5 Solução

Como citado no problema 6.4 esse problema é basicamente uma extensão dele. Precisamos apenas implementar a funcionalidade de atualização. Vale notar que não precisamos mais do tamanho do intervalo o que simplifica um pouco a estrutura:

```
1 void update(int pos, lint value, int no = 1, int l = 0, int r = n) {
2   if (l + 1 == r) {
3     segtree[no].bst = segtree[no].pre = segtree[no].suf = segtree[no].tot
        = v[pos] = value;
4     return;
5   }
6   int mid = (l + r)/2;
7   if (pos < mid)
8     update(pos, value, 2*no, l, mid);
9   else
```

```
10      update(pos, value, 2*no+1, mid, r);
11    segtree[no] = join(segtree[2*no], segtree[2*no+1]);
12  }
```

Sempre que atualizarmos um elemento do vetor, esse elemento é representado por um nó e esse nó tem como total, melhor soma, melhor prefixo e melhor sufixo exatamente o seu valor na posição referente no vetor.

6.7 HORRIBLE - Horrible Queries

6.7.1 URL

https://www.spoj.com/problems/HORRIBLE/

6.7.2 Descrição da entrada

Na primeira linha você receberá T, número de casos de teste. Cada caso de teste começará com N e C. Depois disso, você receberá C:

- 0 p q v - adicionar v para todos os elementos entre p e q inclusive.
- 1 p q - imprimir a soma de todos os elementos entre p e q inclusive.

6.7.3 Descrição da saída

Imprima as respostas das consultas.

6.7.4 Exemplo

Entrada:

```
1
8 6
0 2 4 26
0 4 8 80
0 4 5 20
1 8 8
0 5 7 14
1 4 8
```

Saída:

```
80
508
```

6.7.5 Solução

Aplicação direta da estrutura apresentada no capítulo 5.

Capítulo 7

Conclusão

O desenvolvimento da presente pesquisa possibilitou demonstrar a capacidade e a importância da utilização da estrutura de dados chamada de árvores de segmentos no ambiente de programação competitiva.

De um modo geral, os alunos dedicados à este esporte não têm à sua inteira disposição um vasto e claro aparato acerca do tema, o que motivou a elaboração deste material.

Ao esclarecer com minúcias as possibilidades de utilização da ferramenta, bem como uma das formas de adaptação de implementação da árvores de segmentos, a propagação "preguiçosa", demonstrando o aumento de possibilidades de aplicação da ferramenta e demonstrar os casos práticos de aplicação, foi possível elucidar ao estudante e participante de programação competitiva, quando e como deverá e poderá utilizar a árvores de segmentos na resolução dos problemas.

Tendo em vista a importância do assunto, o desenvolvimento do presente trabalho se deu de maneira desafiadora, diante da necessidade de traduzir o material internacional e o pouco material nacional de uma maneira didática para ser aproveitado aos promissores competidores das maratonas de programação.

Nesse sentido,concluímos a presente análise para que possa ser aproveitada como grande ajuda para os participantes de programação competitiva interessados em conhecer e perscrutar o tema aqui abordado.

Referências Bibliográficas

Bacherikov(2014) Oleksandr Bacherikov. Efficient and easy segment trees. https://codeforces.com/blog/entry/18051, 2014. Citado na pág.

Cormen *et al.***(2009)** Thomas Cormen, Charles Leiserson, Ronald Rivest e Clifford Stein. *Introduction to Algorithms*. The MIT Press, terceira edição. Citado na pág.

Dehghan(2014) AmirMohammad Dehghan. Everything about segment trees. https://codeforces.com/blog/entry/15890, 2014. Citado na pág.

Feofiloff(2008) Paulo Feofiloff. *Algoritmos em linguagem C*. Elsevier. Citado na pág.

Franco(2009) Alvaro Junio Pereira Franco. *Consultas de segmentos em janelas: algoritmos e estruturas de dados*. Tese de Doutorado, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil. Citado na pág.