Árvores de Segmentos (segtrees)

Motivação

Caio é um menino muito cuidadoso com seus carrinhos Rodas Quentes, guardando-os em várias caixinhas que ficam alinhadas em sua estante, uma ao lado da outra. Por ser muito organizado, Caio deixa todas suas n caixinhas numeradas da esquerda para a direita (a primeira é a 1, a segunda a 2, e assim por diante). Em cada caixinha, Caio guarda a_i carrinhos, onde i é o número da i-ésima caixinha.

Vira e mexe, Caio compra mais carinhos Rodas Quentes. E sempre os coloca numa mesma caixinha. Ou seja, se ele fez uma compra de k carrinhos, vai escolher uma caixinha i para colocar esses carrinhos.

Além disso, Caio gosta de brincar com os carrinhos. Mas por ser uma criança muito diversificada, sempre que vai brincar, ele escolhe duas caixinhas, i e j, e pega todos os carrinhos nas caixinhas de i até j para brincar. Depois de ter brincado o suficiente, ele retorna os carrinhos para as caixinhas em que estavam antes (não me pergunte como ele memoriza cada caixinha de cada carrinho).

Apesar de Caio ter uma ótima memória para saber qual carrinho pertence a qual caixinha, ele gostaria de saber com quantos carrinhos ele brincaria se escolhesse duas determinadas caixinhas i e j. Como ele é muito novo e não sabe programar, ele pediu sua ajuda para fazer um programa que ajude ele com isso!

Inicialmente, Caio vai te falar quantas caixinhas ele tem e quantos carrinhos ele possui em cada caixinha. Depois, Caio pode informar duas coisas ao seu programa: 1) ele comprou k carrinhos e os depositou na caixinha i. 2) ele quer saber quantos carrinhos existem da caixinha i até a caixinha j.

Input

A primeira linha contém um valor $n < 10^5$ indicando o número de caixinhas.
A segunda linha contém n valores a_i indicando quantos carinhos estão em cada uma das caixinhas.
A terceira linha contém um valor $q < 10^5$ indicando quantas operações Caio fará com seu programa.
As q linhas seguintes contém operações do seguinte tipo:

- O primeiro valor da linha é um inteiro o indicando o tipo da operação (1 ou 2);

- Se $o = 1$, então a linha conterá dois valores k e i indicando que Caio colocou k carrinhos na caixa i;
- Se $o = 2$, então a linha conterá dois valores i e j ($i \leq j$) indicando que Caio quer saber quantos carrinhos existem entre as caixinhas i e j (incluindo elas);

Obs.: o número de caixinhas não muda.

Output

Toda vez que uma operação do tipo dois for realizada, seu programa deve imprimir um valor numa linha indicando o resultado dessa operação.

Resolvendo o problema ingenuamente

A primeira vista, o problema pode ser bastante fácil e talvez vocês já tenham visto algo do tipo. Entretanto, veremos hoje que esse problema esconde um conceito muito maior por trás: **o de consulta e atualização em segmentos.**

Vejamos então uma resolução imediata do problema:

```
const int N = 112345;
int n;
int arr[N];

void update(int i, int k) { arr[i] += k; }

int query(int i, int j) {
    int ans = 0;
    for (; i <= j; i++) ans += arr[i];
    return ans;
}
```

Esses dois métodos realizam as operações pedidas no enunciado. Além disso são bem fáceis de analizarmos, `update` é $O(1)$ e `query` é $O(n)$. Logo, como, no pior caso, podemos fazer q operações de `query`, temos uma solução $O(nq)$.

Isso passa?

Depende, mas vemos que n e q podem ser da ordem de 10^5, logo teríamos um algoritmo que realiza algo na ordem de 10^{10} operações. E talvez vocês já tenham visto que um computador moderno realiza algo da ordem de 10^9 operações por segundo. Hum... parece então que levaríamos 10 segundos no pior caso.

Isso já nos mostra que, mesmo que uma das operações seja praticamente instantânea ($O(1)$), isso não importa, pois a outra operação ainda nos dá um gargalo de quão ruim nosso programa pode ser.

Além disso, se nós quiséssemos mudar o problema para o seguinte:
Na operação 1), Caio ao invés de colocar os k carrinhos em uma única caixinha, digamos que ele compra sempre vários e vários carrinhos e quer colocar exatamente k em cada uma das caixinha de i até j (inclusas).

Veja que agora as duas operações são $O(n)$!

Será que não podemos diminuir essa complexidade? Veja que estamos assumindo algo muito importante no problema: Caio sempre faz operações considerando caixinhas **adjacentes**. Chamamos isso de **segmentos** de caixinhas.

Existe uma estrutura de dados chamada **Árvore de Segmentos**, que é o tópico da aula de hoje. Ela nos permite realizar as duas operações (que chamamos de consulta e atualização) em $O(\log n)$ (mesmo na segunda versão do problema), muito melhor do que linear como tínhamos.

Representando árvores binárias linearmente

Antes de sair estudando as famosas *segs*, precisamos relembrar e aprender alguns conceitos sobre *árvores binárias*.

Vamos relembrar rapidamente algumas nomenclaturas:

- Uma **árvore** é um grafo acíclico conexo (ou um grafo conexo com $n-1$ arestas);
- Uma **árvore enraizada** é uma árvore na qual fixamos um vértice chamado de **raiz**;
- Numa árvore enraizada, todo nó u possui **filhos** e um **pai**. (Exceto a raiz que não tem pai). O pai e o filhos são os nós adjacentes de u. O pai é o nó adjacente que possui a menor distância até a raiz, é assim que diferenciamos ele dos demais, que são os filhos;
- Um nó u é um **ascendente** de v se u aparece no caminho da raiz até v;
- Um nó u é um **descentente** de v se v aparece no camiho da raiz até u;
- Um nó u é uma **folha** da árvore se u não possui filhos;
- Uma **árvore binária** é uma árvore enraizada na qual todo nó pode ter no máximo dois filhos;
- A **profundidade** de um nó é o número de arestas entre ele e a raiz;
- Uma árvore binária **cheia** é uma árvore binária na qual *todos* os nós possuem dois filhos e todas as folhas possuem mesma profundidade;
- Uma árvore binária **completa** é uma árvore binária cheia exceto talvez o último nível;

Relembrados esse conceitos, existe algo muito importante na computação que é a representação de uma árvore binária *completa* em um array. Provavelmente vocês já sabem como representar grafos e

sabem que costumamos representar usando um desses três métodos:

1. Listas de adjacências;
2. Matriz de adjacências;
3. Lista de arestas.

Entretanto, em se tratando de uma árvore binárias completa (que são usadas em estruturas de dados como a árvore de segmentos e o heap (que vocês verão semestre que vem)), podemos representá-la em um array. Dessa forma, temos acesso em $O(1)$ aos nós dessa árvore (talvez precisando apenas fazer algumas continhas).

Para fazer isso, vamos apenas percorrer a árvore nível por nível e ir numerando ela. A raiz se tornará 1, seu filho esquerdo será 2, seu filho direito será 3, o filho esquerdo do esquerdo da raiz será 4 e assim por diante. (para numerar seria basicamente fazer uma BFS, mas não de fato vamos usar isso no algoritmo da segtree)

Fazendo isso, temos algumas propriedades muito interessante para navegar nessa árvore utilizando o array:

1. A raiz é `seg[1]` ;
2. Seja u um nó qualquer, se eles possuir filhos, então o filho esquerdo é `seg[2i]` e o direito é `seg[2i+1]` .

Assim, temos uma forma já bastante diferente da que estávamos acostumados a trabalhar em grafos e vocês verão que essa forma nos ajudará bastante.

Obs.: Não estamos usando `seg[0]` para nada. Podemos colocar a raiz no 0 ao invés de 1, mas precisaríamos mudar as continhas para achar os filhos de cada nó. (`seg[2i+1]` e `seg[2i+2]` respectivamente para a esq e dir)

Finalmente a árvore de segmentos

A árvore de segmentos é uma árvore binária completa em que cada nó representará um segmento do nosso array original com relação a uma operação (ou várias) que nós escolhemos de acordo com o problema.

Esse valor armazenado no nó se refere à operação `query` . Ou seja, se no nosso problema, queremos saber a soma de i até j, então um nó representando esse segmento vai conter essa soma já pré-calculada para nós.

Para nosso exemplo vamos usar soma, mas a operação poderia ser o mínimo de um segmento, o máximo, a multiplicação, o valor da maior soma contínua possível e por aí vai (as aplicações são

MUITAS).

Assim, nossa árvore vai ter a seguinte *semântica*: a raiz conterá a soma de todo o array $[1, n]$. O seu filho esquerdo vai conter a soma da metade esquerda desse intervalo $[1, n/2]$. O seu filho direito vai conter a soma da metade direita desse intervalo $[n/2 + 1, n]$. E assim funciona para todo nó da árvore, se aquele nó contém a soma do intervalo $[l, r]$, então seu filho esquerdo terá a soma do intervalo $[l, m]$ e o direito do intervalo $[m + 1, r]$, onde $m = (l + r)/2$.

Provavelmente você ainda não entendeu como isso vai deixar as coisas mais rápidas, pois parece que criamos um problema ainda maior. Aparentemente essa árvore contém a soma de todos os segmentos possíveis de construir sobre um array de n elementos e então teríamos que achar um jeito de achar esse nó para responder ao Caio. E, aliás, temos um problema maior ainda: o que eu faço quando Caio comprar mais carrinhos? Como atualizamos essa estrutura louca?

Calma!

Primeiro, essa árvore não tem *todos* os segmentos possíveis de existir. Note que nunca consiguiríamos um nó que represente o segmento $[1, 7]$. Suponha que a raiz representa $[1, 8]$. Então seus filhos representam $[1, 4]$ e $[5, 8]$. E como sempre vamos quebrando esses segmentos ao meio, nunca vamos conseguir um $[1, 7]$. A realidade é que não passam de $n \log n$ nós. Veja que a altura da árvore é sempre proporcional a $\log n$, pois cada nível vai dividindo o intervalo ao meio. Além disso, como cada nó representa um segmento, no "último" nível vamos ter as folhas representando os segmentos $[1, 1], [2, 2], [3, 3]$ e por aí vai. Ou seja, em cada nível temos sempre algo menor ou igual a n, resultando nesse estimador superior. (Se tivéssemos todos os segmentos, teríamos algo quadrático, muito mais do que a árvore contém).

Mas então ainda restam as dúvidas: como fazer aquelas operações nesse novo tipo de estrutura de dados? E como montar essa estrutura a partir do array original?

Construindo

Vamos primeiro mostrar como construir essa árvore partindo do nosso array dado pelo Caio.

```
const int N = 112345;
int st[4*N]; // array representando a seg
int arr[N]; // array dado no problema

void build(int i, int l, int r) {
    if (l == r) { // se estamos numa folha
        st[i] = arr[l]; // então esse nó contém o próprio valor do segmento [l,l]
        // (pois l=r)
    } else {
        int m = (l+r)/2;

        // Construímos recursivamente o filho da esquerda e direita
        build(l, m, 2*i);
        build(m+1,r, 2*i+1);

        // Como o da esquerda possui a soma da metade esquerda do nosso segmento
        // e o da direita possui a soma da metade direita
        //
        // Então o nosso valor é a soma do filho esquerdo com o direito
        st[i] = st[2*i] + st[2*i+1];
    }
}
```

Como estamos falando em árvore, algoritmos recursivos são bastante interativos de pensar. Na construção, nós vamos usar uma recursão com três valores:

1. `i` , indica qual é o nó da árvore que estamos naquele ponto da recursão;
2. `l` , indica a extremidade esquerda do intervalo representado por esse nó;
3. `r` , indica a extremidade direita desse mesmo intervalo.

Resumindo, na chamada `build(i, l, r)` estamos criando o valor do nó i que representa o intervalo $[l, r]$.
(Note que i é um índice referente ao array `st` e que l, r são índices do array `arr`)

Só que vemos que para construir um certo nó i que não seja uma folha, precisamos que seus dois filhos já estejam construídos. Então os construímos recursivamente.

Veja como, na verdade, estamos construindo a árvore de baixo para cima. A sua última camada (a das folhas) será idêntica ao array original, contendo os intervalos $[1, 1], [2, 2]$ e assim por diante. Depois, a camada de cima vai agrupar esses intervalos de dois em dois, contendo os intervalos $[1, 2], [3, 4]$ e assim por diante. Na próxima, teremos os intervalos $[1, 4], [5, 8]$ e assim vai, até chegarmos na raiz, que contém todo o array.

Dessa forma, para construir a árvore toda, basta chamar para a raiz: `build(1, 1, n)` .

Como já vimos, temos um máximo de $n \log n$ nós e, portanto, a construção será $O(n \log n)$.

Atualizando

Vamos agora responder à pergunta de como fazer a operação de atualização. A atualização também é bastante simples. Se queremos atualizar um valor i do array original, então precisamos nos perguntar quais são os nós que representam intervalos que contêm i e atualizar todos esses segmentos.

Imediatamente, temos que a raiz contém esse i, pois ela é a soma de todo o array. Agora, como os filhos da raiz dividem esse intervalo em dois, sabemos que i com certeza está em exatamente um desses dois intervalos. Para achar esse intervalo, basta saber se i é menor ou igual que m ou se é maior. Se for menor ou igual, então o intervalo da esquerda contém i, caso contrário, será o intervalo da direita que o contém.

E vamos repetindo isso até encontrar a folha que representa o intervalo $[i, i]$.

```
void update(int i, int l, int r, int pos, int k) {
    if (l == r) { // Se chegamos na folha, então pos = l = r
        st[l] += k; // Ao chegarmos na raiz, somamos k diretamente
    } else {
        int m = (l+r)/2;

        // Se a posição que queremos atualizar está para a esquerda,
        if (pos <= m) update(2*i, l, m, pos, k); // atualizamos a esquerda.
        // Se a posição que queremos atualizar está para a direita,
        else update(2*i+1, m+1, r, pos, k); // atualizamos a direita.

        // Após atualizar um dos filhos, atualizamos o nó atual
        st[i] = st[2*i] + st[2*i+1];
    }
}
```

Vemos que, se o problema mudasse para "*Caio vende k carrinhos*" ou qualquer outra operação, poderíamos simplesmente trocar o `+= k` por `-= k`.

Além disso, podemos observar que esse método é chamado para todos os nós no caminho da raiz até a folha que representa $[i, i]$. Logo, o número de nós é no máximo a altura da árvore, que é $O(\log n)$.

Parece que tivemos uma perda, não é verdade? Pois antes tínhamos um algoritmo $O(1)$ para fazer isso e agora temos um algoritmo $O(\log n)$. Mas se lembre da extensão do problema que propus no começo.

Por enquanto, veja que estamos fazendo um update apenas em um ponto do array (o que chamamos de **point update**). Se quisermos mudar o update para que possamos fazer update em um segmento

$[i, j]$ (o que chamamos de **range update**), podemos usar essa mesma estrutura de dados para fazer isso também em $O(\log n)$, algo que não conseguíamos fazer somente com o array. Veremos como fazer isso mais adiante.

Consulta

Para fazermos a segunda operação, precisamos agora entender como utilizar essa estrutura pré-processada para realizar isso de forma rápida.

Primeiro, como já discutimos, a árvore não possui todos os segmentos possíveis. Logo, o que precisaremos fazer é somar segmentos para conseguir o segmento que desejamos.

Por exemplo, se quisermos a soma de $[1, 7]$, então vamos pegar os segmentos $[1, 4]$, $[5, 6]$ e $[7, 7]$. Somando os três nós que representam esses segmentos, teremos a soma do segmento representado por $[1, 7]$.

Dessa forma, apresentemos o algoritmo para depois o discutir:

```
int query(int i, int l, int r, int ql, int qr) {
    // Se o segmento da consulta [ql, qr] não intersecta o segmento atual [l, r]
    if (qr < l || r < ql) return 0; // Retornamos uma soma nula

    // Se o segmento atual está completamente contido no segmento da consulta
    if (ql <= l && r <= qr) return st[i]; // Retornamos a soma de l a r

    // Caso contrário, então o segmento atual possui apenas parte da soma desejada
    // e devemos quebrar esse nó nos seus dois subintervalos
    int m = (l+r)/2;
    return query(2*i, l, m, ql, qr)+query(2*i+1, m+1, r, ql, qr);
}
```

Vamos entender o que ele está fazendo.
Novamente temos as mesmas semânticas de sempre para `i`, `l` e `r`.

1. `ql` é a esquerda do intervalo da consulta;
2. `qr` é a direita do intervalo da consulta;

Ou seja, se Caio quer quantos carrinhos existem da caixinha 1 até a 7, imprimimos `query(1, 1, n, 1, 7)`.

Sempre começamos a busca pela raiz, pois ela representa o intervalo inteiro e depois vamos "filtrando" esse intervalo até ele ser exatamente o que queremos.

O algoritmo separa os casos em três:

1. $[l, r]$ e $[ql, ql]$ não possuem intersecção. Isso significa que, por exemplo, chegamos ao nós representando $[3, 4]$ e queremos a soma de 5 até 7. Como não precisamos da soma desse intervalo, podemos simplesmente parar por aqui e retornar 0 nessa parte da busca, não indo mais fundo na árvore (não olhamos os intervalos $[3, 3]$ ou $[4, 4]$);
2. $[l, r]$ está dentro de $[ql, qr]$. Seria o caso de estarmos buscando a soma de 1 até 7 e chegarmos no intervalo $[1, 4]$. Sabemos que não precisamos ir mais a fundo, pois esse intervalo já contém uma parte da soma que desejamos.
3. O intervalo $[l, r]$ contém $[ql, qr]$ e algo mais. Seria o caso de estar procurando a soma de 1 a 7 e estivéssemos na raiz $[1, 8]$. Vemos que não podemos retornar o valor da raiz, pois estaríamos incluindo a soma da oitava caxinha. Assim, o que fazemos é retornar a soma da query da esquerda com a query da direita.

Vamos simular rapidamente `query(1, 1, 8, 1, 7)` encaixando cada chamada em um dos três acima:

1. `query(1, 1, 8, 1, 7)` -> caso 3, chamamos para os filhos (primeiro esquerda);
2. `query(2, 1, 4, 1, 7)` -> caso 2, retornando a soma de $[1, 4]$;
3. Fim da recursão esquerda de 1., chamando a direita:
4. `query(3, 5, 8, 1, 7)` -> caso 3, chamamos para os filhos (primeiro esquerda);
5. `query(6, 5, 6, 1, 7)` -> caso 2, retornando a soma de $[5, 6]$;
6. Fim da recursão esquerda de 4., chamando a direita:
7. `query(7, 7, 8, 1, 7)` -> caso 3, chamamos para os filhos (primeiro esquerda);
8. `query(14, 7, 7, 1, 7)` -> caso 2, retornando a soma de $[7, 7]$;
9. Fim da recursão esquerda de 7., chamando a direita:
10. `query(15, 8, 8, 1, 7)` -> caso 1, retornando 0;
11. Final da recursão direita de 7., somando $[7, 7]$ com 0 dá a soma de $[7, 7]$;
12. Final da recursão direita de 4., somando $[5, 6]$ com $[7, 7]$ dá a soma de $[5, 7]$;
13. Final da recursão direita de 1., somando $[1, 4]$ com $[5, 7]$ dá a soma de $[1, 7]$,

como desejávamos.

Para esse caso em específico, parece que estamos fazendo mais trabalho até do que somar diretamente os 7 valores como fazíamos antes, mas a realidade é que para árvores muito grandes, os casos 1. e 2. são uma otimização realmente boa, fazendo com que esse algoritmo seja também $O(\log n)$ (a prova é um pouco mais difícil).

Análise

Como já vimos, por mais que a operação `update` tenha se saído prejudicada, no total, prejudicar ela para fazer com que a `query` caia para $O(\log n)$ é um grande avanço, pois agora nosso algoritmo é

$O(q \log n)$.

Vimos que a construção da árvore é $O(n \log n)$, então a realidade é que nosso algoritmo fará o problema em $O(q \log n + n \log n)$, pois primero lemos o array e construímos a árvore e depois realizamos as operações.

No nosso caso, como q pode ser tão grande quanto n, teríamos um algoritmo $O(n \log n)$. Pode ser que alguns problemas mais simples não precisem de segtrees, seja lá porque o q é muito pequeno ou por algum outro motivo. Mas como não podemos garantir que os problemas vão sempre ser fáceis, dominar segs é muito interessante, pois nos permite realizar operações antes lineares em tempo logarítmico.

Lazy Propagation

Agora vamos voltar ao problema mencionado na parte de atualização:
Suponha que, na operação 1), Caio ao invés de colocar os k carrinhos em uma única caixinha, ele quer colocar exatamente k em cada uma das caixinha de i até j (inclusas).

Agora, como já mencionamos, temos um problema em que as duas operações atuam sobre segmentos (tanto o update como a query agora são do tipo **range**).

Árvore de segmentos conseguem fazer as duas operações em $O(\log n)$, apenas com um pouco mais de trabalho (ou de preguiça).

A técnica chama-se **lazy propagation** (propagação preguiçosa).

Bom, já vimos que o algoritmo usado pela consulta basicamente vai cortando o intervalo até termos exatamente o intervalo que precisávamos, certo? E quando o temos, paramos de descer na árvore, pois já temos toda a informação que precisamos (se chegamos em $[1, 4]$, não precisamos descer mais para $[1, 2]$, $[3, 4]$ ou, pior, para $[1, 1]$, $[2, 2]$, $[3, 3]$, $[4, 4]$).

A pergunta então é, *será que dá para fazer o mesmo com a atualização*?

A resposta é *mais ou menos*.

Bom, evidemente, se queremos atualizar o intervalo $[1, 5]$ e o algoritmo chega em $[1, 4]$, então, se só atualizarmos $[1, 4]$, pode ser que alguma consulta posterior precise da informação contida no nó que representa $[1, 2]$ (filho de $[1, 4]$) e, como apenas atualizamos seu pai, este nó não estará atualizado e nos levará a uma resposta errada!

Mas puxa, eu sou um programador preguiçoso, então eu vou deixar um aviso pro *eu* do futuro: "Ei, eu atualizei aqui o $[1, 4]$ e ainda não atualizei os filhos dele ($[1, 2]$ e $[3, 4]$)".

Depois, quando o *eu* do futuro for fazer uma consulta que passa por $[1, 2]$, ele vai falar "opa, esse nó tá desatualizado daquela vez lá que eu anotei, então deixa eu atualizar ele primeiro e depois usar a informação atualizada". (e, como somos preguiçosos, vamos apenas atualizar ele e marcar seus filhos)

A mesma coisa aconteceria se eu precisasse de, por exemplo $[3, 3]$, que é um descentente de $[1, 4]$. Nós deixamos anotado no $[3, 4]$ que ainda falta atualizar ele, e sabemos que, para chegar no $[3, 3]$, nós precisamos passar por $[3, 4]$ primeiro. Logo, ao passar por $[3, 4]$, nós vamos verificar que ele está desatualizado. Mas, novamente, sou muito preguiçoso, então eu vou atualizar ele, por que estou nele agora, mas eu vou de novo passar pros filhos dele: "Ei, falta atualizar $[3, 3]$ e $[4, 4]$" e nós vamos atualizar esses nós apenas quando estivermos neles. Daí, logo em seguida eu acabaria indo para o $[3, 3]$ e acabaria fazendo a atualização nele.

Ou seja, essa ténica não muda apenas a atualização, mas também a consulta. Nós vamos usar a famosa técnica de deixar tudo em cima da hora para fazer. Ou seja, na atualização, assim que pudermos parar, nós paramos e avisamos que os filhos precisam ser atualizados.
E quando que atualizamos eles? Apenas quando acabarmos passando por eles.

Vamos ao código:

```
const int N = 112345;
int st[4*N]; // array representando a seg
int lz[4*N]; // array representando as ataulizações pendentes
int arr[N]; // array dado no problema

void build(int i, int l, int r) {
    if (l == r) {
        st[i] = arr[l];
    } else {
        int m = (l+r)/2;

        build(l, m, 2*i);
        build(m+1,r, 2*i+1);

        st[i] = st[2*i] + st[2*i+1];
    }

    // A única mudança desse método:
    // Não temos nenhuma mudança em nenhum nó inicialmente
    lz[i] = 0;
}

// Um método novo que verifica se há mudanças pendentes
// se sim, ele faz a mudança no nó atual e empurra o "lembrete" de atualizações
// para os filhos
void push(int i, int l, int r) {
    // Se temos mudanças pendentes
    if (lz[i] != 0) {
        // Atualizamos (vamos discutir essa fórmula depois)
        st[i] += lz[i]*(r-l+1);

        if (l != r) { // se não é uma folha
            // passamos a pendência para os filhos
            lz[2*i] += lz[i];
            lz[2*i+1] += lz[i];
        }

        // Como fizemos a atualização, não há mais pendências
        lz[i] = 0;
    }
}

void update(int i, int l, int r, int ql, int qr, int k) {
    // Se tinha alguma mudança pra ser feita, fazemos ela antes de mais nada
    push(i, l, r);

    if (qr < l || r < ql) return;
    if (ql <= l && r <= qr) {
        // Se chegamos num ponto que podemos parar, vamos atualizar só
        // esse nó e marcar nossos filhos.
```

```
        // O jeito mais rápido (de codar) é somar k no lz[i] e chamar o push
        lz[i] += k;
        push(i, l, r);
        return;
    }

    int m = (l+r)/2;
    st[i] = st[2*i] + st[2*i+1];
}


int query(int i, int l, int r, int ql, int qr) {
    // A única diferença na query é que devemos, sempre que visitamos um nó,
    // chegar se há alguma mudança pendente e, se houver, atualizar antes de
    // mais nada.
    push(i, l, r);

    if (qr < l || r < ql) return 0;

    if (ql <= l && r <= qr) return st[i];

    int m = (l+r)/2;
    return query(2*i, l, m, ql, qr)+query(2*i+1, m+1, r, ql, qr);
}
```

Vemos que as mudanças não foram tantas:

- Vamos guardar as mudanças pendentes num array `lz`;
- Vamos inicializar esse array com um valor nulo em relação à operação da atualização;
- Criamos uma função nova `push` que verifica se há mudanças e, se houver, as faz, passando essas pendências para os filhos;
- Chamamos essa função sempre que visitamos um nó (seja quando estamos atualizando ou consultando);
- Também usamos essa função para fazer a atualização de forma preguiçosa.

Discutamos agora a fórmula `lz[i]*(r-l+1)`:

Para entender o porquê dessa fórmula, lembre-se que estamos atualizando o intervalo $[l, r]$ e colocando $lz[i]$ carrinhos em cada caixa. Logo, para saber quantos carrinhos colocamos no total, da caixinha l até a r, basta multiplicar o número de carrinhos que colocamos em cada uma $(lz[i])$ pelo número de caixinhas $(r - l + 1)$. É daí que essa fórmula sai.

Se o seu problema utiliza alguma outra operação diferente, você terá que pensar como você atualiza o segmento $[l, r]$ de acordo com a operação do seu problema.

Notas finais

Algumas nomenclaturas que eu gostaria de fixar:

A primeira seg que mostrei é uma seg de **point update** e **range query**.

A segunda, utilizando lazy propagation, é uma de **range update** e **range query**.

Também existe uma seg de **range update** e **point query**. Implementação e explicação pode ser encontrada aqui.

Se você tiver um problema com **point update** e **point query**, então não há motivo para usar segs.

Após isso, vale a pena entender como podemos generalizar as operações que as segs usam.

Utilizei soma no update e na query, mas poderíamos por exemplo, no update, utilizar a operação de deixar com exatamente k carrinhos, vender k carrinhos, somar k carrinhos $\mod 12$, multiplicar a quantidade atual por k e assim vai.

Na query, poderíamos querer qual caixinha tem o menor número de carrinhos, ou saber qual é esse menor número de carrinhos (talvez as duas coisas ao mesmo tempo). Poderíamos também querer a multiplicação de todas as caixinhas do segmento e assim vai.

Além disso, podemos ter operações muito mais complexas que não são essas matemáticas que estamos acostumados, nos problemas da lista, vocês encontraram esse tipo de coisa.

As operações da consulta, por exemplo, pode ser não comutativas. Ou seja, unir o intervalo $[l, m]$ com $[m+1, r]$ resulta num valor diferente de unir o intervalo $[m+1, r]$ com $[l, m]$. Nesse caso, talvez o problema queira o mínimo ou o máximo desses valores (ou qualquer outro requisito proposto) e você terá que fazer as duas uniões para depois verificar qual atende o requisito.

Além disso, vale a pena lembrar que a nossa seg está guardando inteiros, mas poderíamos querer guardar qualquer tipo de coisa. Tudo dependerá do problema. Pode ser que tenhamos que guardar caracteres, talvez strings inteiras, talvez arrays ou matrizes de inteiros, pars, sets, structs das mais diversas! (o vídeo que passei acima mostra, no final da aula, uns dois tipos de segs assim)

Quero apenas abrir um pouco a sua cabeça para mostrar que esse assunto que vimos resolve esse problema que parece besta (não deixe o Caio saber que falei isso), mas na realidade, segment trees são estruturas de dados poderosíssimas!

Por fim, quero agradecer a leitura. Espero que ela tenha sido proveitosa e você tenha gostado tanto dessa estrutura de dados como eu gosto.

Qualquer sugestão, comentário, crítica, erro que encontrou no texto etc., pode falar com o autor: Lucas Paiolla Forastiere (handle: Giatro).