Seletiva IOI 2021

- MinQueue
- Histograma

André Amaral de Sousa

Autor

Sobre

- Ex-Maratonista (ICPC 2013)
- Ex-Olímpico (IOI 2012)
- Problem Setter na Brasileira 2019
- Engenheiro de Computação (POLI-USP)
- Professor no colégio ETAPA
- Contato:
  - andremfq@gmail.com

MinQueue

MinQueue

Definição

Estrutura que suporta

- Inserir elemento no final
- Remover elemento do início
- Obter o menor elemento ativo na estrutura

MinQueue

Versão 1 - Usando MinStacks

Vamos pensar primeiro como implementar uma MinStack

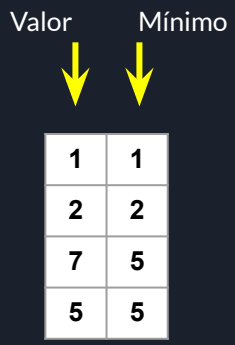
Estrutura que suporta

- Inserir elemento no topo
- Remover elemento do topo
- Obter o menor elemento ativo na estrutura

MinQueue

Versão 1 - Usando MinStacks

Basta fazer uma stack (pilha) e além de guardar o valor, guardar também o menor elemento dali para baixo:



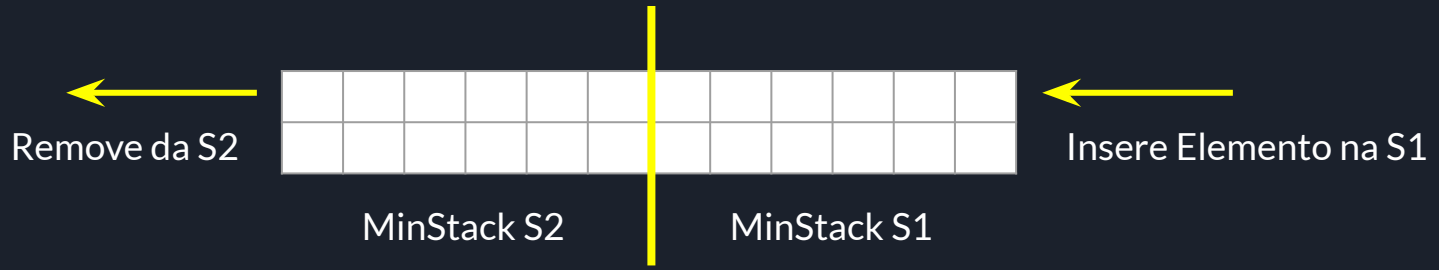
Insere Elemento:

Ao inserir um novo elemento calcula o mínimo entre ele e o mínimo no topo

MinQueue

Versão 1 - Usando MinStacks

Agora que sabemos construir uma MinStack, basta criar uma queue com 2 stacks



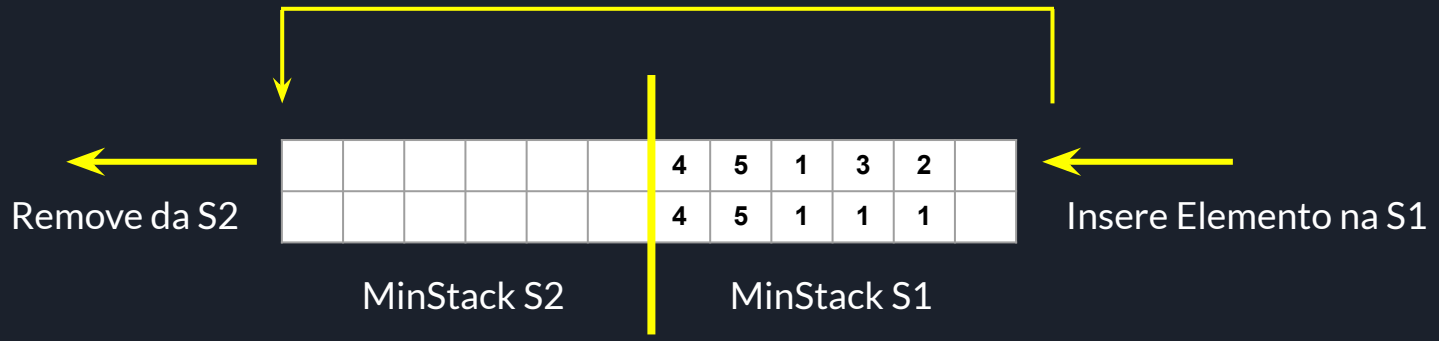
Mínimo: é o mínimo entre o mínimo das duas MinStacks

Mas e quando S2 estiver vazia ? Removermos de onde ?

MinQueue

Versão 1 - Usando MinStacks

Quando S2 estiver vazia, retira TODOS os elementos de S1 e coloca em S2, note que inverterá a ordem deles.

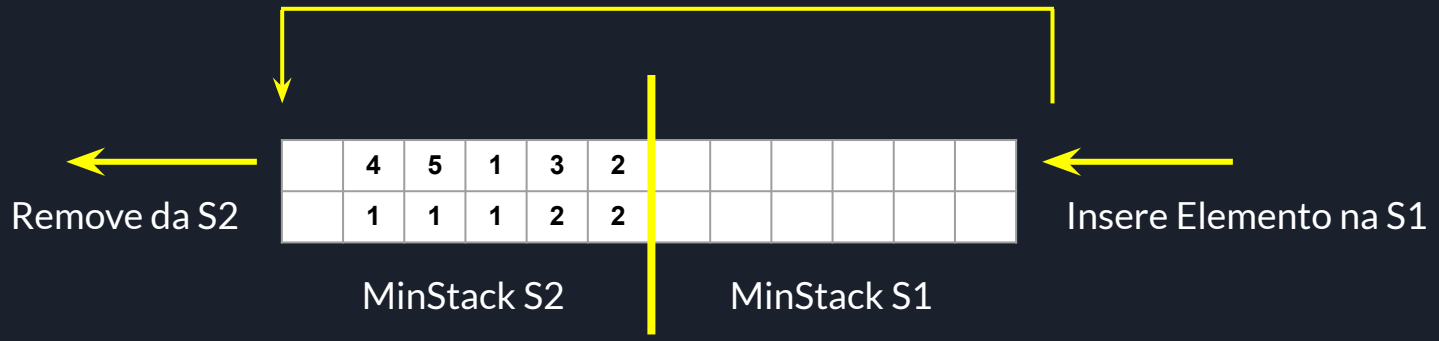


Mínimo: é o mínimo entre o mínimo das duas MinStacks

MinQueue

Versão 1 - Usando MinStacks

Quando S2 estiver vazia, retira TODOS os elementos de S1 e coloca em S2, note que inverterá a ordem deles.



Mínimo: é o mínimo entre o mínimo das duas MinStacks

MinQueue

Versão 1 - Usando MinStacks

Perceba que a complexidade é O(1) para todas as operações

Na verdade a operação de Remoção é Amortizado O(1), ou seja, uma operação de remoção é O(N) já que todos podem passar de S1 para S2, mas todas as operações de remoção juntas também são O(N) pois cada elemento só pode ir de S1 para S2 uma única vez.

MinQueue

Versão 2 - Usando Deque

Outra forma de pensar é manter na estrutura somente os elementos que ainda podem ser mínimo

Ex.: 2 3 7 5

Quando o 5 entra na estrutura, o 7 nunca mais poderá ser o mínimo, pois o 5 aparece depois então será removido depois que o 7, então sempre que o 7 estiver na estrutura o 5 também estará, logo o 7 é inútil para o cálculo, e podemos removê-lo.

Portanto quando chega um elemento removemos todos os elementos maiores que ele e que estejam no final da fila. Note que precisamos de uma estrutura que suporte remover do fim também (além de remover do início por conta da instrução de remover), por isso usaremos a Deque (Double Ended Queue) que insere e remove tanto do início quanto do final.

MinQueue

Versão 2 - Usando Deque

Mas se simplesmente jogarmos os elementos fora, teremos um problema ao remover

Ex.: 2 3 7 5 1 9

Neste exemplo apenas os elementos 1 e 9 estarão na nossa estrutura (o 1 arrancou todos os anteriores quando entrou na estrutura). Se então após colocarmos todos os elementos, removermos um elemento da estrutura iremos remover o 1, quando na realidade deveríamos remover o elemento 2

Como consertar ?

MinQueue

Versão 2 - Usando Deque

Mas se simplesmente jogarmos os elementos fora, teremos um problema ao remover

Ex.: 2 3 7 5 1 9

Neste exemplo apenas os elementos 1 e 9 estarão na nossa estrutura (o 1 arrancou todos os anteriores quando entrou na estrutura). Se então após colocarmos todos os elementos, removermos um elemento da estrutura iremos remover o 1, quando na realidade deveríamos remover o elemento 2

Basta manter para cada elemento o índice em que ele entrou na estrutura, e manter dois índices informando quais índices estão ativos na estrutura.

Ex.: valor 2 3 7 5 1 9 ini = 1 e fim = 6 Ao remover vemos se o primeiro elemento tem índice igual ao ini, se sim removemos ele, e sempre aumentamos o ini

Índice 1 2 3 4 5 6

|----------|

inúteis

MinQueue

Versão 2 - Usando Deque

```
struct MinQueue {
  deque<pair<int, int> > d;
  int ini, fim;

  MinQueue() { ini = 1; fim = 0; }

  void push(int v) {
    while(!d.empty() && d.back().first >= v) d.pop_back();

    d.push_back(make_pair(v , ++fim));
  }

  void pop() {
    if(!d.empty() && d.front().second == ini++)
      d.pop_front();
  }

  int getMin() {
    return d.front().first ;
  }
};
```

MinQueue

Exercício

Pense em como fazer uma MinQueue que também suporte a seguinte operação:

- Add(val) - Some o valor val em todos os elementos ativos da Minqueue

Ex.: 2 3 7 5 1 9
Add(3)
5 6 10 8 4 12
Insere(7)
5 6 10 8 4 12 7
Remove() -> retira o 5
6 10 8 4 12 7
Add(2)
8 12 10 6 14 9

MinQueue

Exercício - Solução

Basta manter uma variável soma, dizendo quanto devemos somar em todos os elementos ativos na estrutura.

Mas isso dá problema com os novos elementos que chegarem depois dessa instrução de soma, pois podemos somar em elementos que não deveríamos.

Para lidar com este erro, basta ao invés de inserir o elemento v, inserir o elemento v - soma

MinQueue

Exercício - Solução

Ex.: 2 3 7 5 1 9 soma = 0
Add(3)
2 3 7 5 1 9 soma = 3
Insere(7)
2 3 7 5 1 9 4 soma = 3
Remove() -> retira o 2 + 3 = 5
3 7 5 1 9 4 soma = 3
Add(2)
3 7 5 1 9 4 soma = 5

MinQueue

Exercício - Solução (TEM UM ERRO, ENCONTRE)

```
struct MinQueue {
  deque<pair<int, int> > d;
  int ini, fim, soma;

  MinQueue() { ini = 1; fim = 0; soma = 0; }

  void push(int v) {
    while(!d.empty() && d.back().first >= v) d.pop_back();

    d.push_back(make_pair(v - soma , ++fim));
  }

  void pop() {
    if(!d.empty() && d.front().second == ini++)
      d.pop_front();
  }

  void add(int val) {
    soma += val;
  }

  int getMin() {
    return d.front().first + soma ;
  }
};
```

MinQueue

Exercício - Solução (TEM UM ERRO, ENCONTRE)

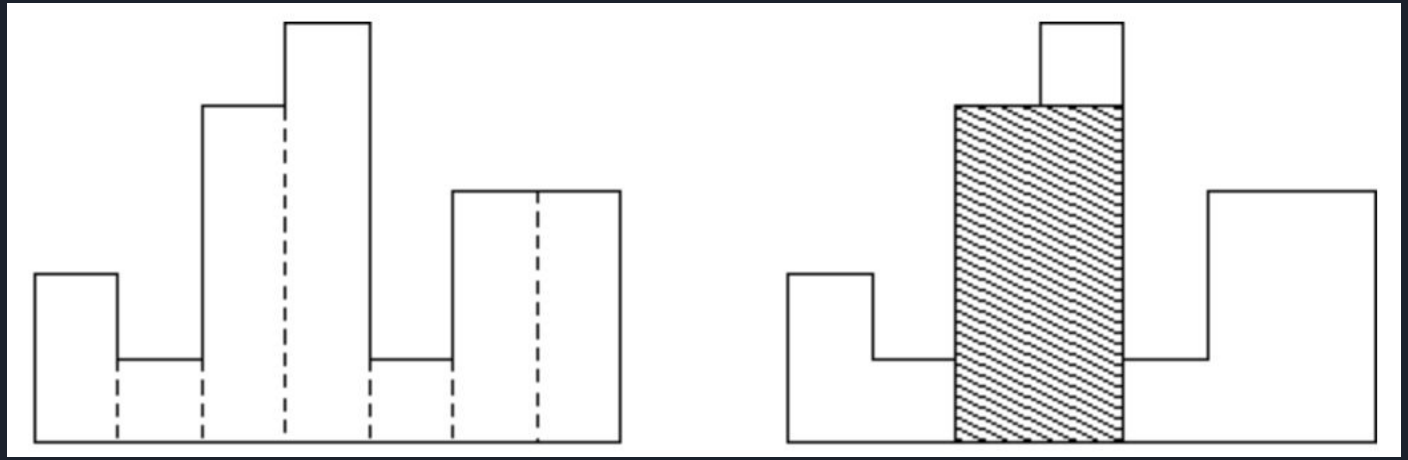
```
struct MinQueue {
  deque<pair<int, int> > d;
  int ini, fim, soma;

  MinQueue() { ini = 1; fim = 0; soma = 0; }

  void push(int v) {
    while(!d.empty() && d.back().first >= v) d.pop_back();

    d.push_back(make_pair(v - soma , ++fim));
  }

  void pop() {
    if(!d.empty() && d.front().second == ini++)
      d.pop_front();
  }

  void add(int val) {
    soma += val;
  }

  int getMin() {
    return d.front().first + soma ;
  }
};
```

MinQueue

Lista de Exercícios

Aquatic Surf
Sequência
Pilots
Desk Ordering
Trous de loup
Little Bird
Lucky Country

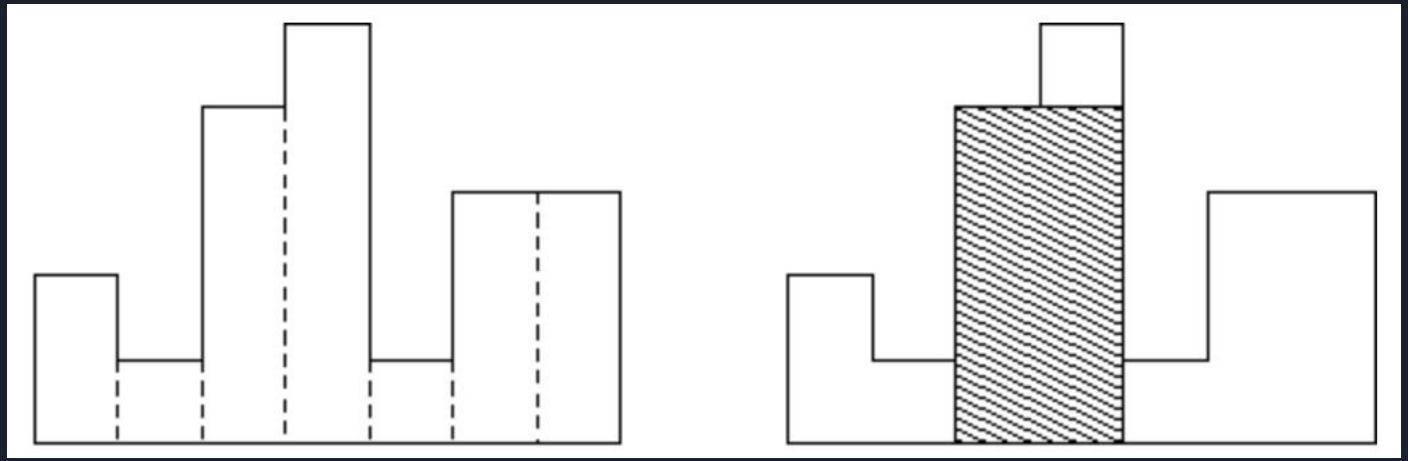
Histograma



Histograma

Problema

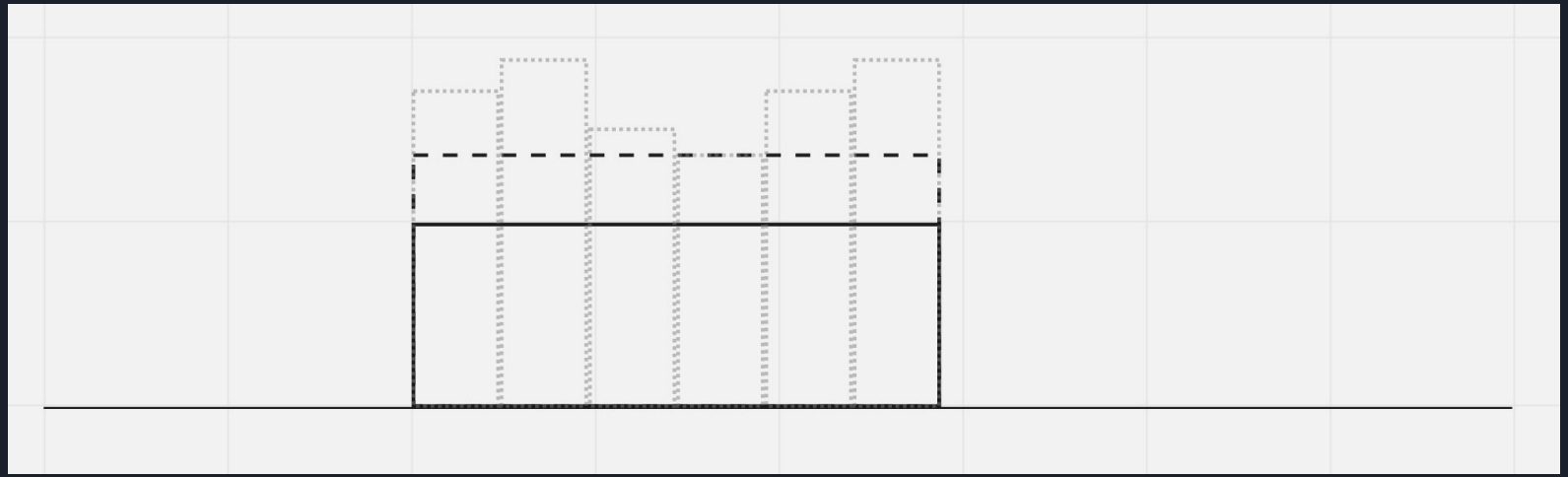
Dado um número N e N barras de largura 1 e altura H[i] (as alturas são dadas); Em outras palavras dado um Histograma, encontre a área do retângulo de maior área que cabe dentro deste histograma.



Histograma

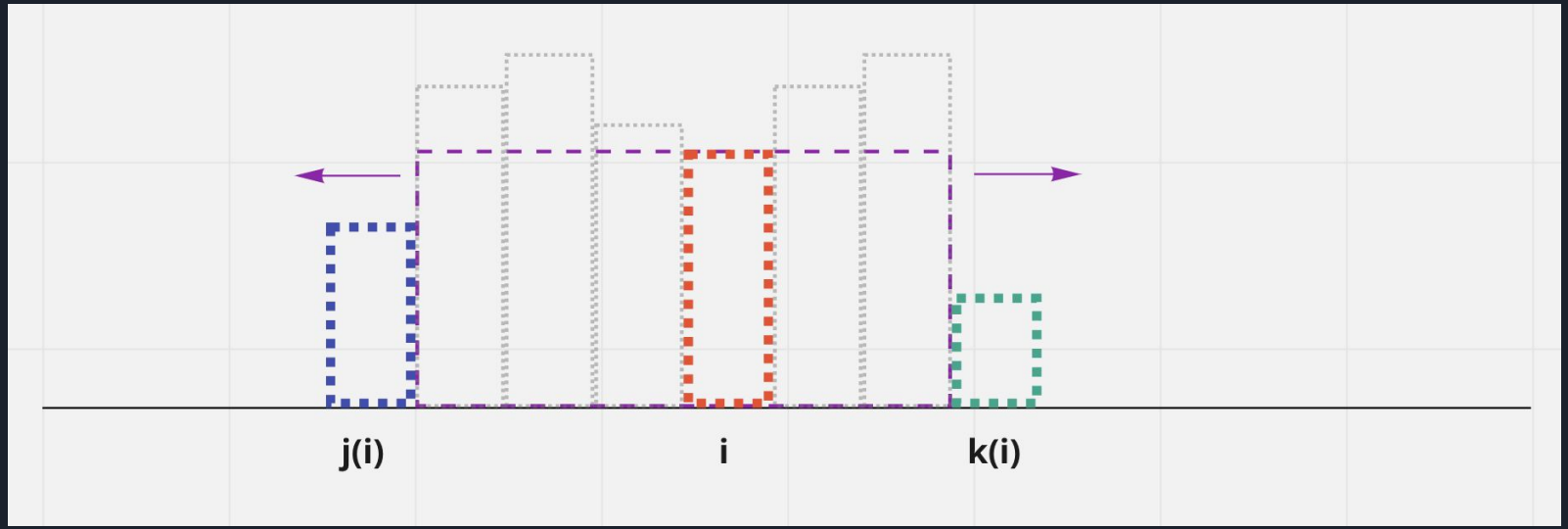
Observação

A resposta ótima sempre tem a altura de uma das barras (pois senão daria para aumentar a resposta, conforme figura abaixo)



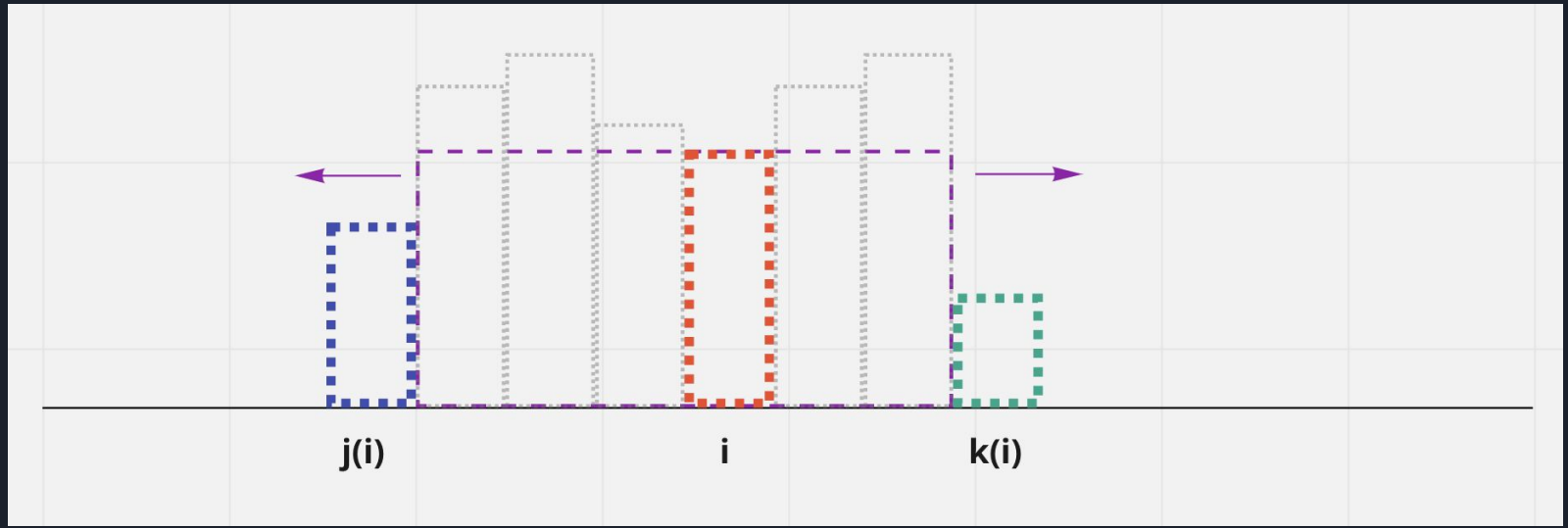
Histograma

Portanto podemos encontrar para cada barra qual é o maior retângulo que tem sua altura e passa por ele, e a resposta ótima será o maior entre todos.



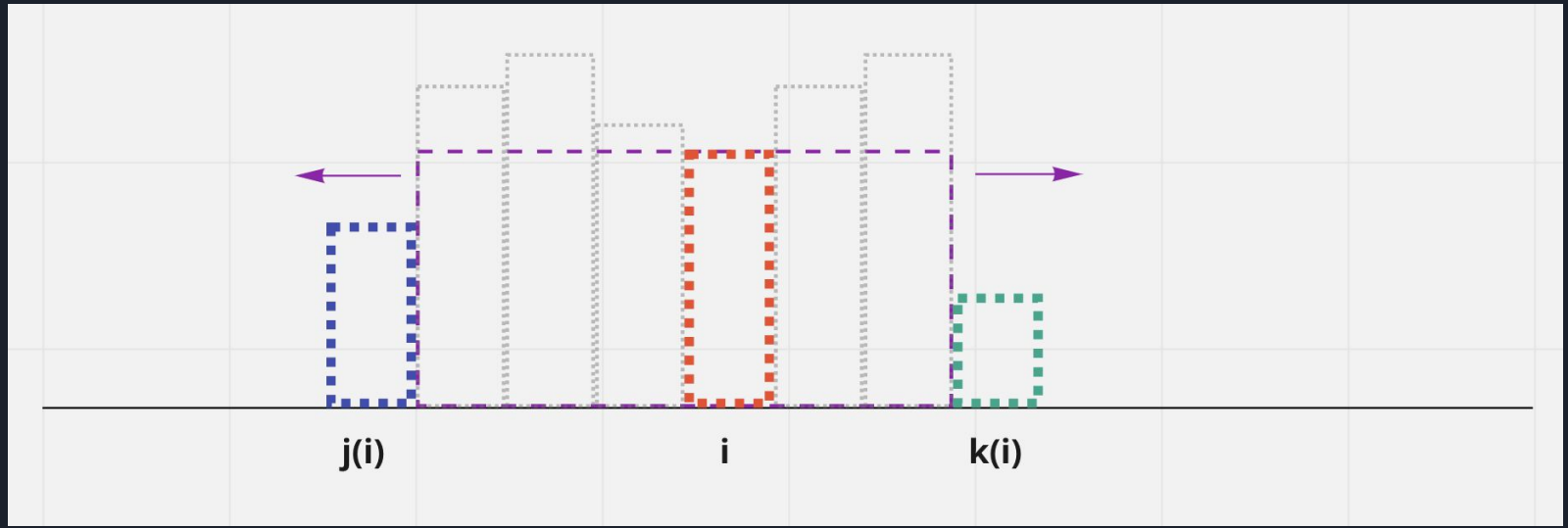
Histograma

Para encontrar o melhor retângulo relativo a barra i temos que esticar ao máximo cada um dos lados (pois a altura já está fixada, então para maximizar a área, a partir da barra i, esticamos o máximo para a direita e o máximo para a esquerda).



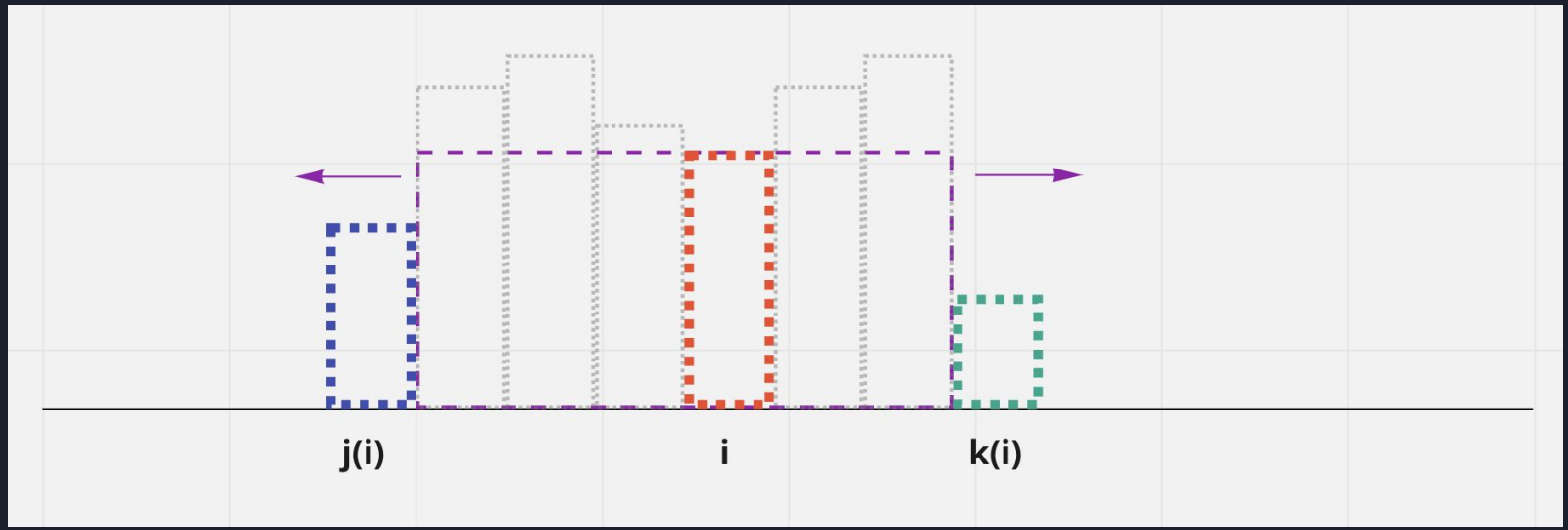
Histograma

Seja k(i) o menor índice maior que i com altura menor que H[i] (k(i) > i, H[k(i)] < H[i]), em outras palavras k(i) é o primeiro índice à direita de i com altura menor que a dele, ou seja o processo de esticar pela direita chegará até k(i) - 1



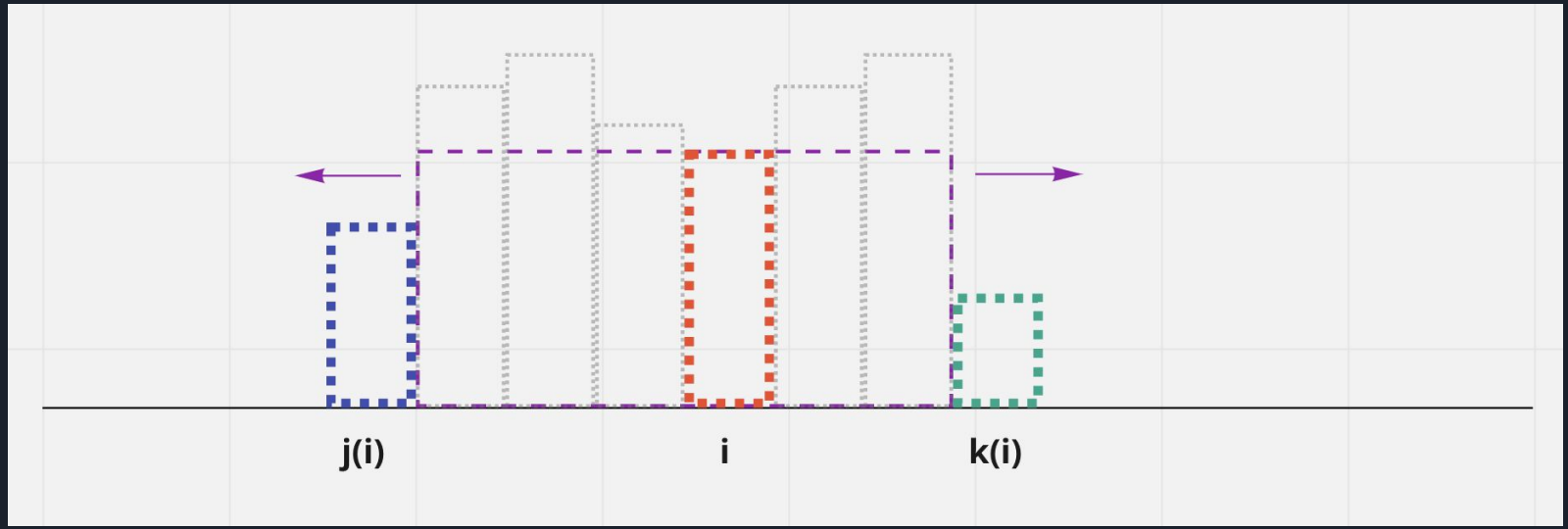
Histograma

De forma análoga seja j(i) o maior índice menor que i com altura menor que H[i] (j(i) < i, H[j(i)] < H[i]), em outras palavras j(i) é o primeiro índice à esquerda de i com altura menor que a dele, ou seja o processo de esticar pela esquerda chegará até j(i) + 1



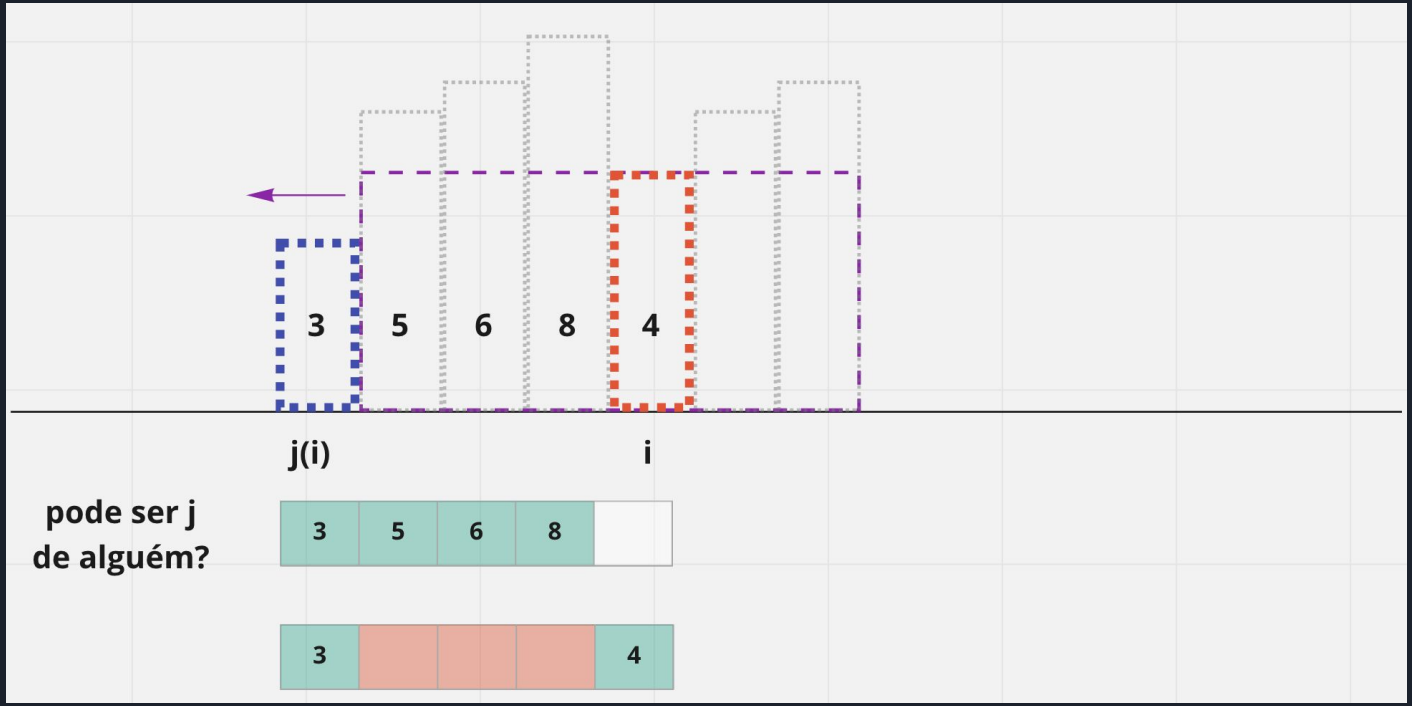
Histograma

Para finalizar o problema precisamos encontrar k(i) e j(i) para todo i. Uma forma é simular o processo de esticar em cada i ou seja de fato ir percorrendo as barras partindo de i para a direita até achar a primeira menor, e o mesmo para a esquerda. Perceba que este procedimento de esticar é O(n) e assim a complexidade final ficará O(n^2)



Histograma

Vamos focar primeiro em encontrar j(i) para todo i.



Histograma

Observe o exemplo anterior, 3 5 6 8 4. Imagine antes de processar o 4 (3 5 6 8 x) vamos marcar quais barras podem ser o j de alguma outra barra (a primeira menor a esquerda):

- O 3 pode ser a menor a esquerda se x for 4 por exemplo (x representa a próxima barra, imagine agora que não sabemos que será o valor 4).
- O 5 pode ser a menor a esquerda se x for 6.
- O 6 também pode se x for 7.
- O 8 também pode ser x for maior ou igual a 9.

Então notem que neste momento todos podem ser o menor a esquerda de alguma outra barra que está por vir.

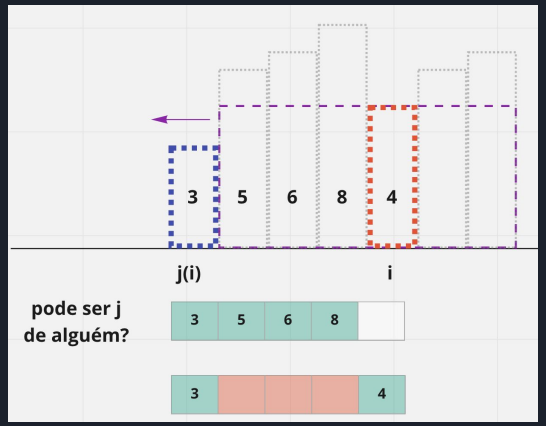
Agora Imagine quando chega o 4 (3 5 6 8 4 x). O 8 não pode mais ser a menor a esquerda de ninguém, pois como 4 é menor que 8 e a barra em questão terá que estar a direita do 4, o 4 aparecerá antes do 8. Portanto podemos descartar o 8. Note que o mesmo ocorre com o 6 e o 5, pois também são maiores que o 4 e estão à esquerda do 4.

Histograma

Perceba que as alturas das barras que ainda podem ser j de outra formam uma sequência crescente, pois quando chega a altura x ela inutiliza todos os maiores que x.

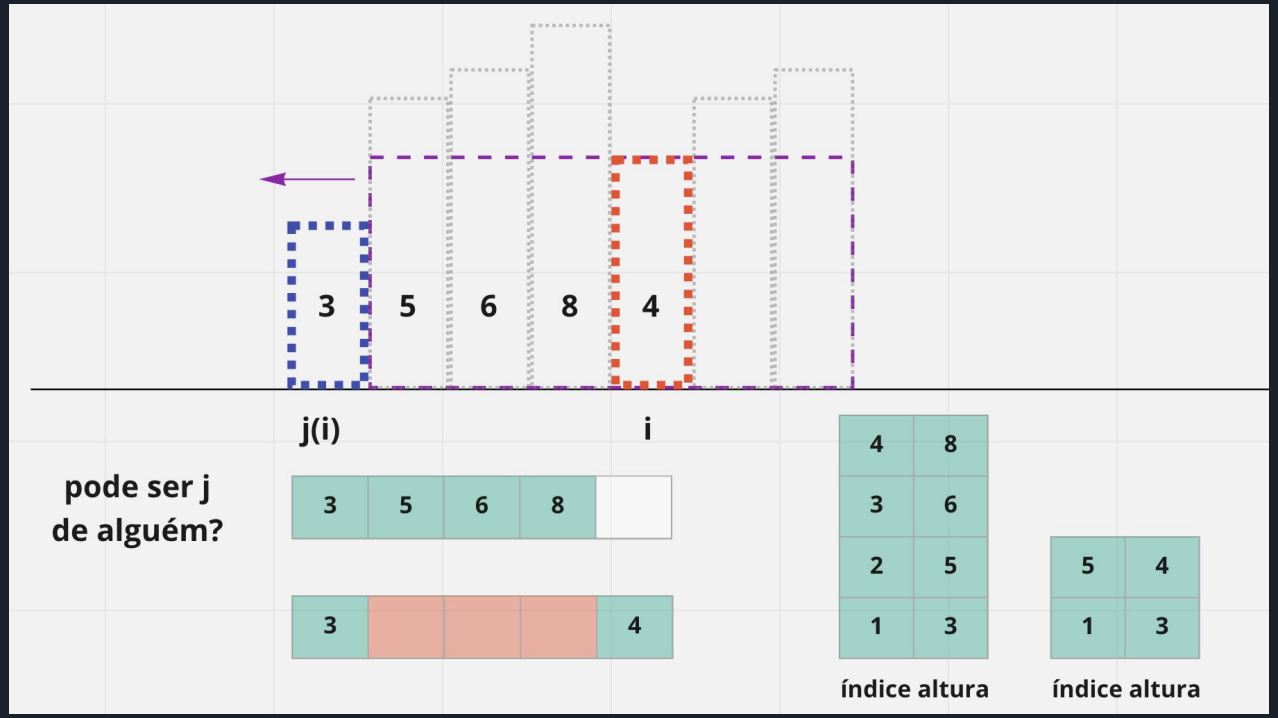
Precisamos de uma estrutura para manter de forma eficiente as barras que ainda podem ser j de outra. Note que ao chegar uma altura x como ela inutiliza todos os maiores que x e a sequência é não decrescente então ela só inutiliza os últimos adicionados.

Podemos utilizar uma pilha para manter as barras que ainda podem ser j de outra



Histograma

Podemos utilizar uma pilha para manter as barras que ainda podem ser j de outra



Histograma

Podemos utilizar uma pilha para manter as barras que ainda podem ser j de outra

Iremos manter na pilha o índice da barra (com isso é possível obter a altura através do vetor H, mas para visualizar melhor coloquei a altura na pilha também). Ao adicionar uma barra, retiramos do topo todas as maiores que a barra atual, e então adicionamos a barra atual no topo da pilha. Perceba que logo antes de adicionar a barra atual no topo, a barra que estiver no topo será a j da barra atual.

Histograma

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100010;
int h[MAXN], j[MAXN];
stack<int> pilha;
int main() {
    int n; scanf("%d", &n);
    for(int i = 1; i <= n; i++) scanf("%d", &h[i]);
    h[0] = -1; pilha.push(0);
    for(int i = 1; i <= n; i++) {
        while(h[pilha.top()] > h[i]) pilha.pop();
        j[i] = pilha.top();
        pilha.push(i);
    }
    return 0;
}
```

*errinho: while(h[pilha.top()] >= h[i]) pilha.pop(); pois queremos o primeiro estritamente menor

Histograma

Quanto fica a complexidade ?

Como temos um while dentro do for a princípio podemos pensar que ficará O(n^2) porém perceba que cada barra só será inserida uma única vez na pilha e só pode ser retirada da pilha uma única vez, portanto a complexidade final fica O(n)

***para inserir uma barra na pilha fica O(n) mas para inserir todas também fica O(n), dizemos que para inserir uma barra fica O(1) amortizado.**

Histograma

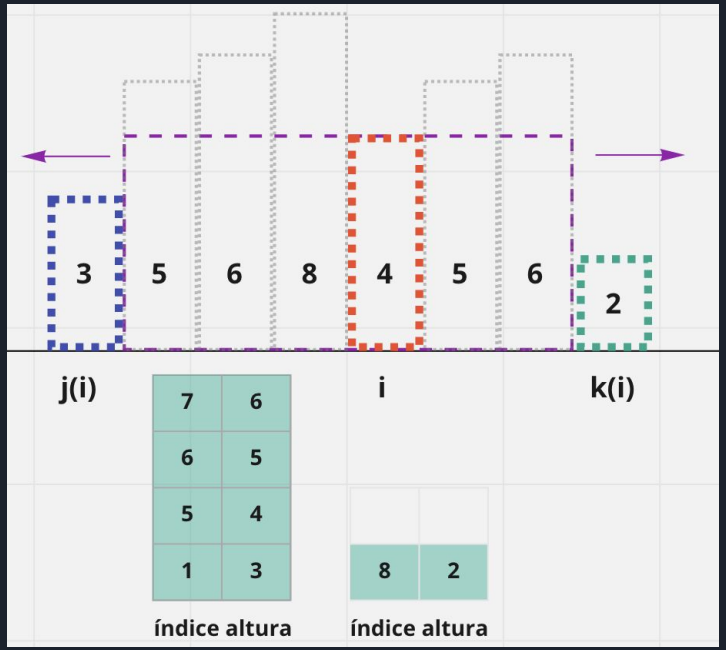
Como finalizar ?

Podemos calcular k(i) de forma análoga (passando de trás pra frente no vetor)

Observação:

a barra que irá retirar a barra i da pilha será k(i).

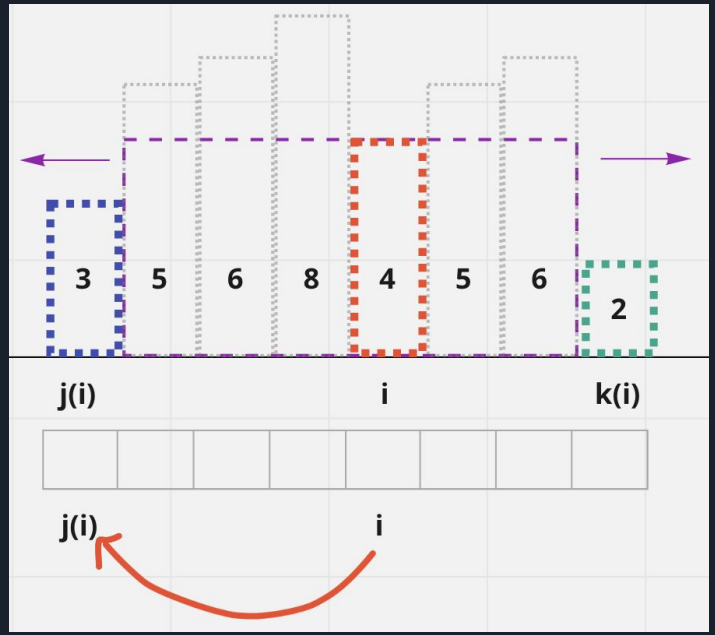
Portanto durante o cálculo de j(i) também conseguimos calcular k(i).



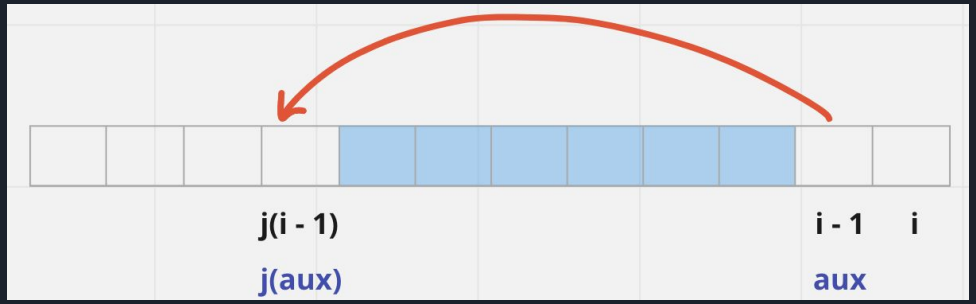
Histograma

Outra Solução

Vamos visualizar o problema de forma diferente, visualize uma seta de cada índice i para o j(i)



Histograma



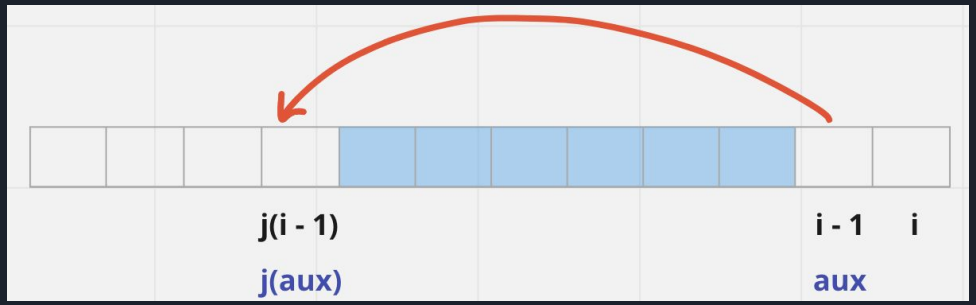
Suponha que já determinamos j (a seta) pra todo índice menor que i

Se H[i-1] < H[i] então já encontramos o primeiro menor a esquerda, e assim j(i) = i - 1

Senão teremos que procurar algum índice menor que i - 1 para encontrar quem é o j(i). Porém perceba que todos os valores entre os índices de j(i - 1) a i - 1 (marcados em azul na figura) são maiores que H[i], pois sabemos neste caso que H[i - 1] >= H[i] e todo valor entre j(i - 1) e i - 1 é maior que H[i - 1]. Portanto neste caso j(i) será no máximo j(i - 1).

Podemos usar esta mesma ideia pro j(i - 1), ou seja, testa se ele é menor que H[i] se for encontramos o j(i), senão podemos pular para j(j(i - 1)), pelo mesmo argumento.

Histograma



Assim para determinar o j(i) podemos fazer:

```
int aux = i - 1;
while(h[aux] >= h[i]) aux = j[aux];
j[i] = aux;
```

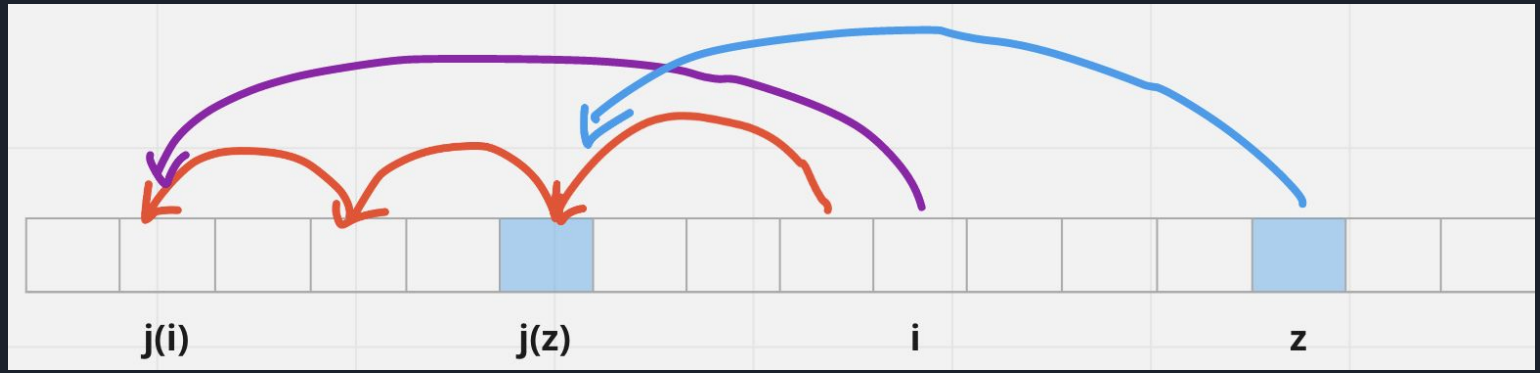
Histograma

Desta forma chegamos na seguinte solução:

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100010;
int h[MAXN], j[MAXN];
int main() {
    int n; scanf("%d", &n);
    for(int i = 1; i <= n; i++) scanf("%d", &h[i]);
    h[0] = -1;
    for(int i = 1; i <= n; i++) {
        int aux = i - 1;
        while(h[aux] >= h[i]) aux = j[aux];
        j[i] = aux;
    }
    return 0;
}
```

Histograma

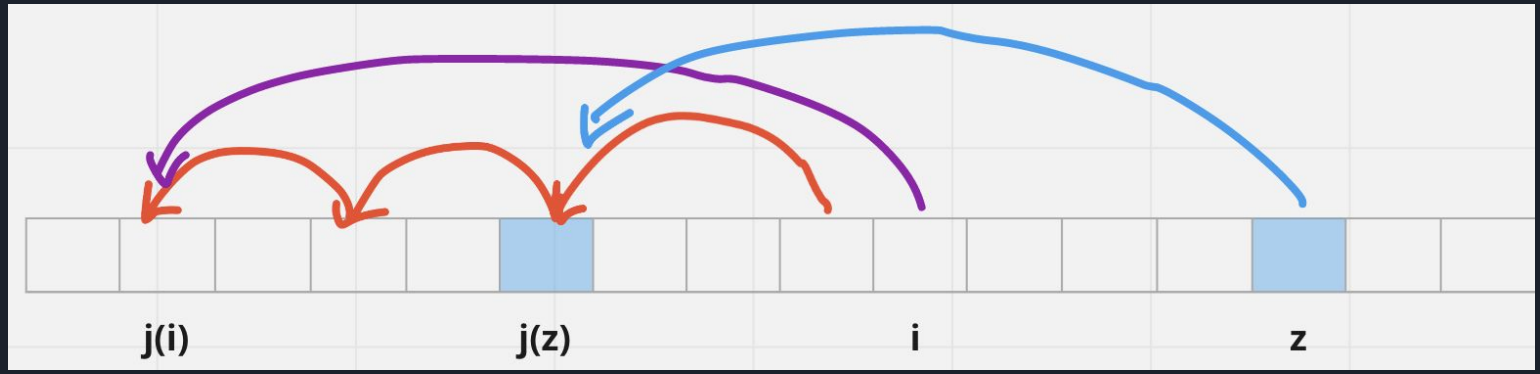
Mas qual será a complexidade desta solução ? Novamente parece O(n^2), mas...



Observação: Se para calcular j(i) "pulamos" pela seta de um índice, nenhum outro índice passará por ela. Note que Pular por uma seta equivale a ter uma iteração no while. Visualmente essa afirmação equivale a dizer que duas setas não se cruzam, pois para para passar novamente por uma seta seria preciso cruzar (conforme a figura).

Histograma

Mas qual será a complexidade desta solução ? Novamente parece O(n^2), mas...



Esta observação é verdadeira pois, suponha um índice z > i cuja seta cruze a seta do i, ou seja, j(i) < j(z) < i isso significa que o primeiro valor menor que H[z] é H[j(z)] mas acontece que H[i] <= H[j(z)] e j(z) < i < z, o que é absurdo.

Portanto cada seta só pode ser percorrida uma única vez, então a quantidade total de iterações pelo while é O(n), portanto a complexidade final é O(n)

Histograma

Como finalizar ? Podemos calcular k(i) de forma análoga (passando de trás pra frente no vetor)

Observação: o índice que irá pular pela seta do i será k(i)

Por definição k(i) é o primeiro índice a direita de i com valor menor ou igual que H[i], portanto j(k(i)) <= j(i) ou seja o primeiro valor menor que H[k(i)] está num índice menor ou igual a j(i). Como mostrado anteriormente, nenhuma seta se cruza e para todo índice z entre i e k(i) temos que j(z) >= i. Portanto juntando todos estes fatos, quando formos calcular j(k(i)) necessariamente teremos que passar pela seta do i

Histograma

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100010;
int h[MAXN], j[MAXN], k[MAXN];
int main() {
    int n; scanf("%d", &n);
    for(int i = 1; i <= n; i++) scanf("%d", &h[i]);
    h[0] = -2; h[n + 1] = -1;
    for(int i = 1; i <= n + 1; i++) {
        int aux = i - 1;
        while(h[aux] >= h[i]) {
            k[aux] = i;
            aux = j[aux];
        }
        j[i] = aux;
    }
    return 0;
}
```

Histograma

Na verdade as duas soluções são análogas. A grande questão é que j(i) é exatamente o índice logo abaixo na pilha, então para passar pela pilha não precisamos armazená-la basta usar o j. Note também que ao processar o índice i - 1 ao final sempre o colocamos na pilha, portanto quando chegamos no índice i o i - 1 está no topo.

Histograma

Lista de Exercícios

HISTOGRA
CTGAME
Matriz super-legal
Minimum Sum