Otimizando o front-end

Uma breve introdução sobre complexidade de tempo e estruturas de dados

Gustavo Oliveira

Alguns dados...

- 97% dos brasileiros conectados já realizaram compras pela internet;
- 64% dos usuários tendem a deixar a página e buscar outro site que seja mais rápido;
- 79% dos usuários que compram online com frequência, não compram em sites que são muito lentos.

https://www.hostgator.com.br/blog/site-lento-como-resolver/

Redação E-Commerce Brasil

Lojas virtuais perdem R$ 132,05 milhões durante Black Friday e Cyber Monday

Quinta-feira, 05 de dezembro de 2019 • BLACK FRIDAY Tempo de leitura: 10 minutos •

https://www.ecommercebrasil.com.br/noticias/instabilidade-e-commerce-na-black-friday

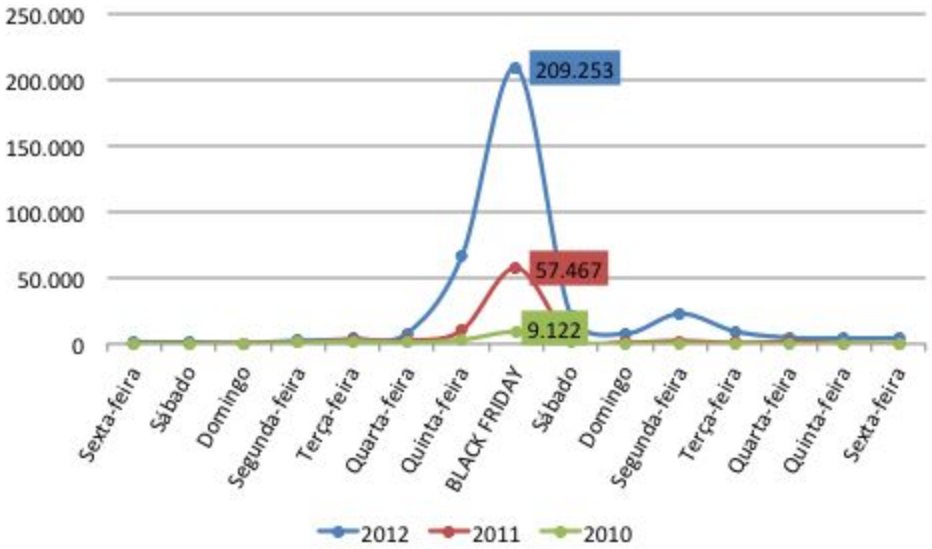
Início » Brasil » Queixas na Black Friday 2020 crescem 45%, segundo Reclame Aqui

Queixas na Black Friday 2020 crescem 45%, segundo Reclame Aqui

Desde quarta-feira (25), Reclame Aqui registrou média de 115 reclamações por hora relacionadas à Black Friday 2020

https://tecnoblog.net/388683/queixas-na-black-friday-2020-crescem-45-segundo-reclame-aqui

Cenário na black friday



http://blog.hiplatform.com/monitore-em-tempo-real-as-mencoes-digitais-da-black-friday/

Podemos ter degradação na performance em todas as frentes: infra/front/back

SEO

O Google disse 2 segundos de carregamento dos sites. E agora?

BY RAPHAEL LASSANCE - 13 DE JUNHO DE 2016 - 11 MINS READ

Mas quando o assunto é o tempo de espera do usuário, só estamos falando de carregamento da homepage?

Site lento: 7 pontos que devem ser avaliados

1. Testar a velocidade na conexão padrão
2. Analisar os dados do site
3. Verificar HTML/CSS
4. Executar JavaScript
5. Testar a usabilidade em dispositivos móveis
6. Dimensionar o conteúdo para mobile e verificar a legibilidade do texto
7. Comparar a velocidade do seu site com outros semelhantes

https://www.hostgator.com.br/blog/site-lento-como-resolver/

Site lento: 7 pontos que devem ser avaliados

1. Testar a velocidade na conexão padrão
2. Analisar os dados do site
3. Verificar HTML/CSS
4. Executar JavaScript
5. Testar a usabilidade em dispositivos móveis
6. Dimensionar o conteúdo para mobile e verificar a legibilidade do texto
7. Comparar a velocidade do seu site com outros semelhantes

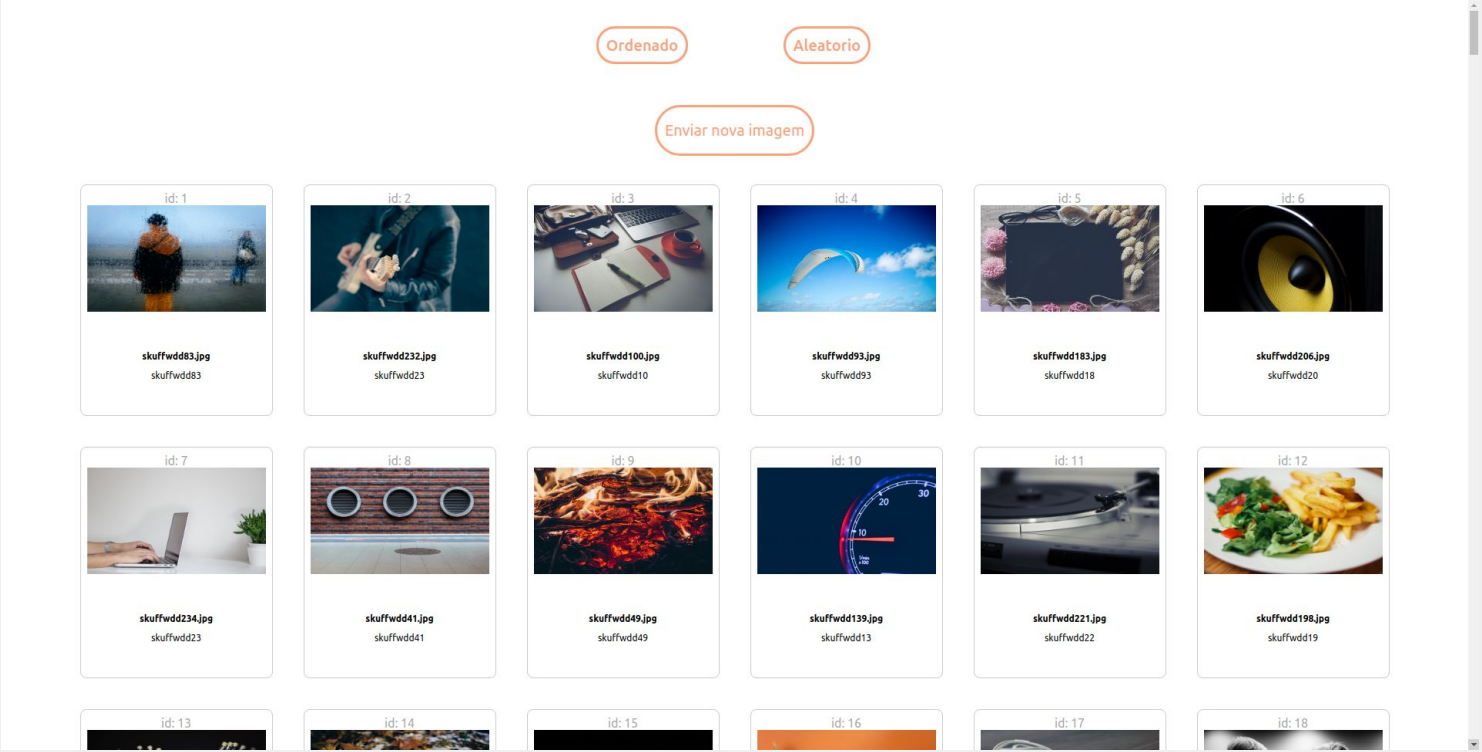
https://www.hostgator.com.br/blog/site-lento-como-resolver/

Vamos ver alguns cases...

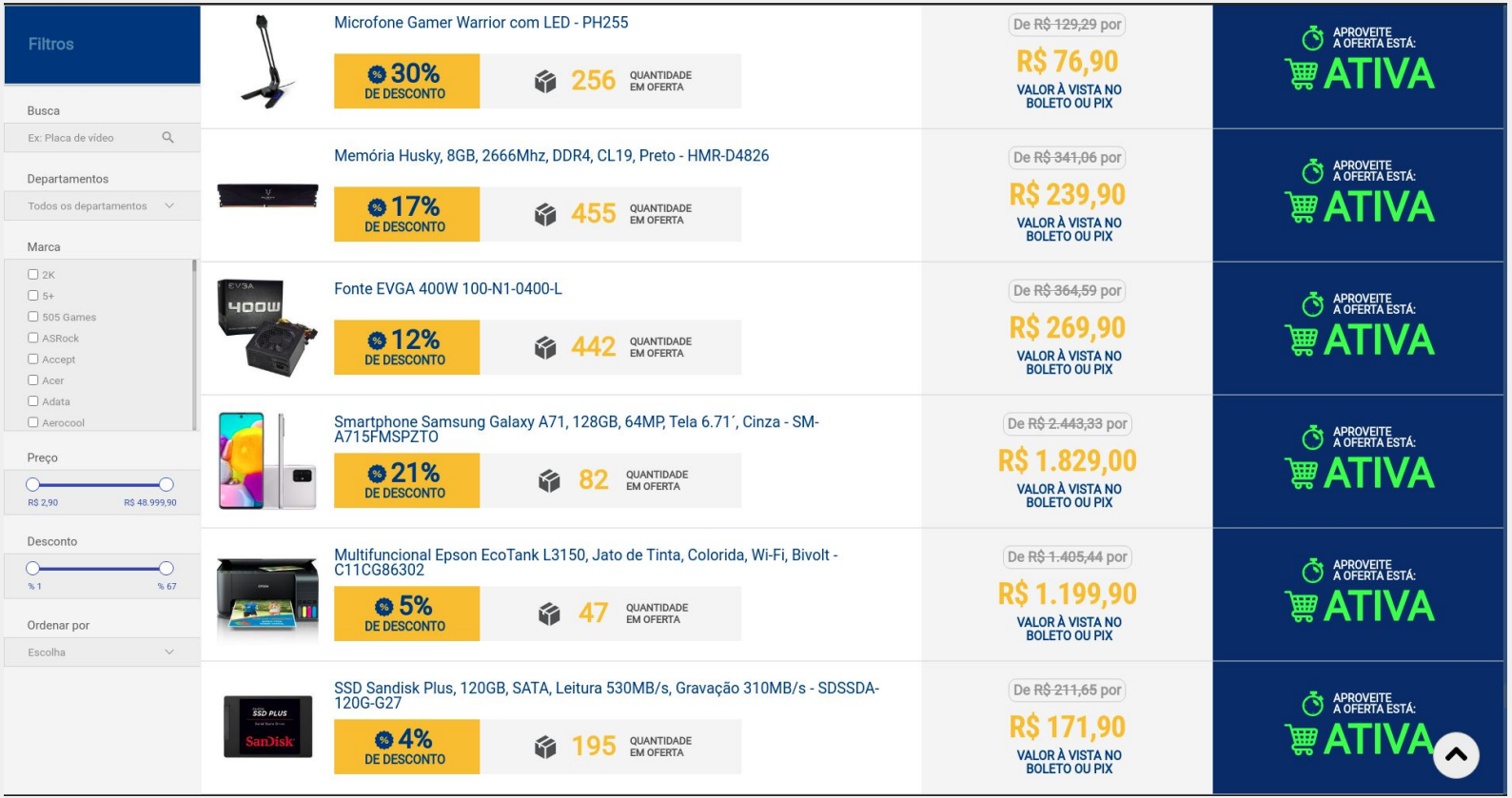
Use case 1

Use case: o usuário deseja poder pesquisar o card pelo id.

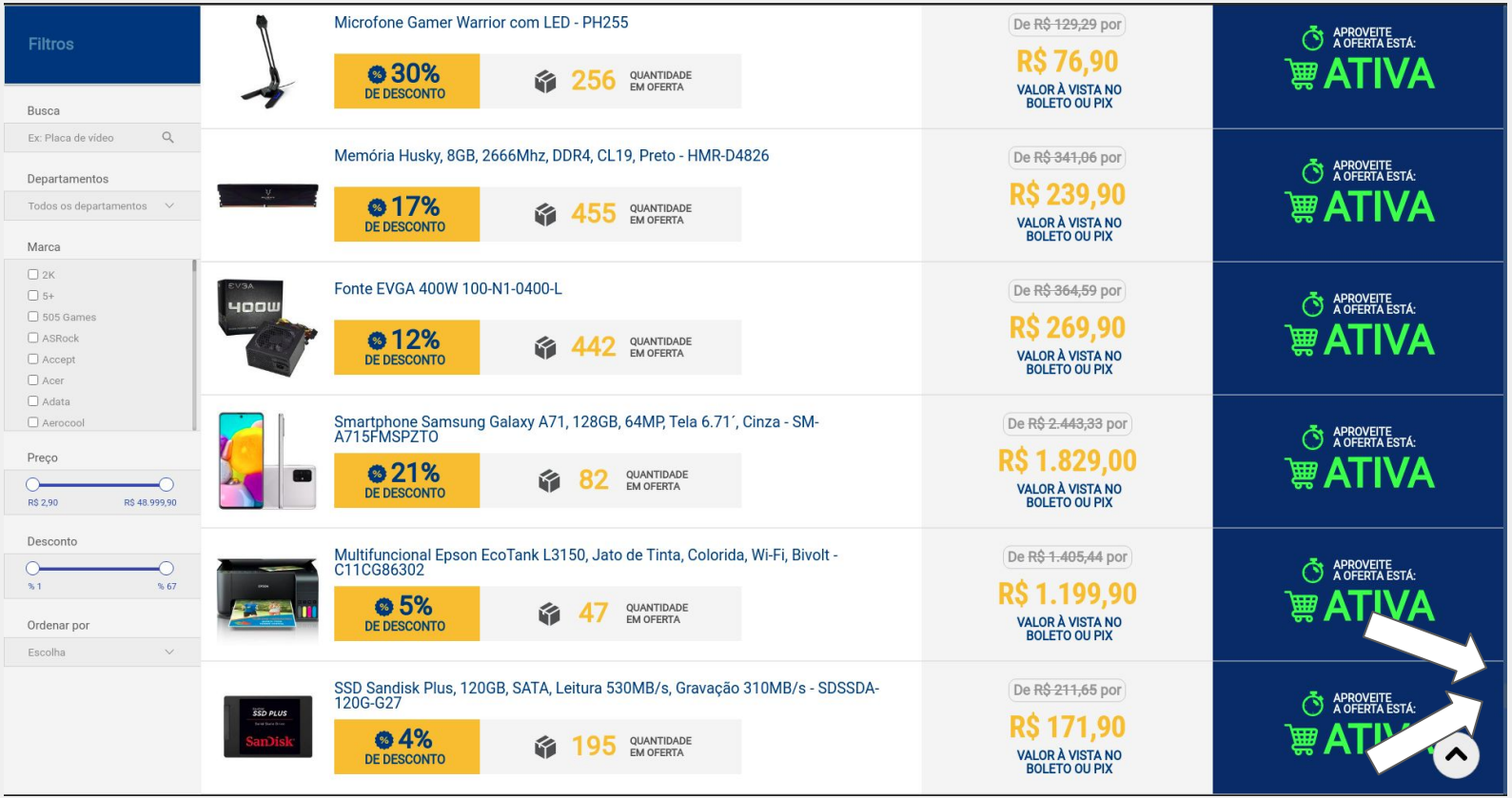
*É garantido que os cards estão ordenados pelo id. Por baixo dos panos, o que o front precisa fazer é retornar o índice do id no array (se existir).



Exemplo de site com muitos elementos carregados

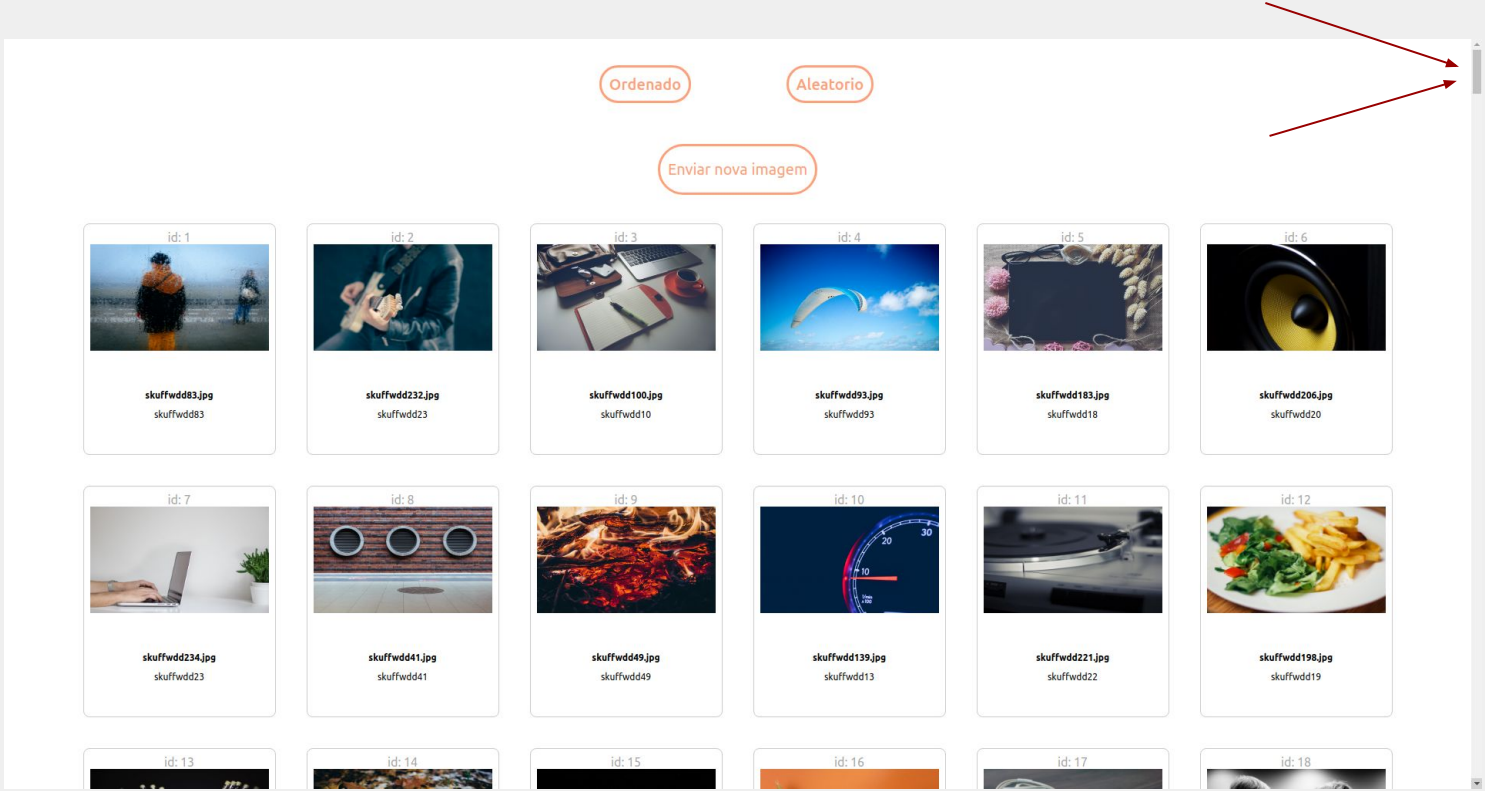


Exemplo de site com muitos elementos carregados



Use case: o usuário deseja poder pesquisar o card pelo id.

*É garantido que os cards estão ordenados pelo id. Por baixo dos panos, o que o front precisa fazer é retornar o índice do id no array (se existir).



Exemplo

Dado o array de ids: [1, 4, 6, 9];

O usuário deseja pesquisar o id 1 no array; return → 0

O usuário deseja pesquisar o id 3 no array; return → false (ou -1)

O usuário deseja pesquisar o id 9 no array; return → 3

Busca sequencial

```
1 function sequentialSearch(arr, id) {
2   for (let index = 0; index < arr.length; index++)
3     if (arr[index] === id)
4       return index;
5
6   return false;
7 }
```

Complexidade: O(*n*) (lê-se “Ó de n”)

```
1 function sequentialSearch(arr, id) {
2   for (let index = 0; index < arr.length; index++)
3     if (arr[index] === id)
4       return index;
5
6   return false;
7 }
```

Ó de quê? Uu qui eh içó?

Ok… Precisamos dar um passo para trás...

Complexidade de algoritmos

Complexidade de algoritmos

Definição informal: Tem a ver com quanto tempo e memória esse algoritmo gasta de acordo com o tamanho de sua entrada e o quanto o algoritmo consome da CPU.

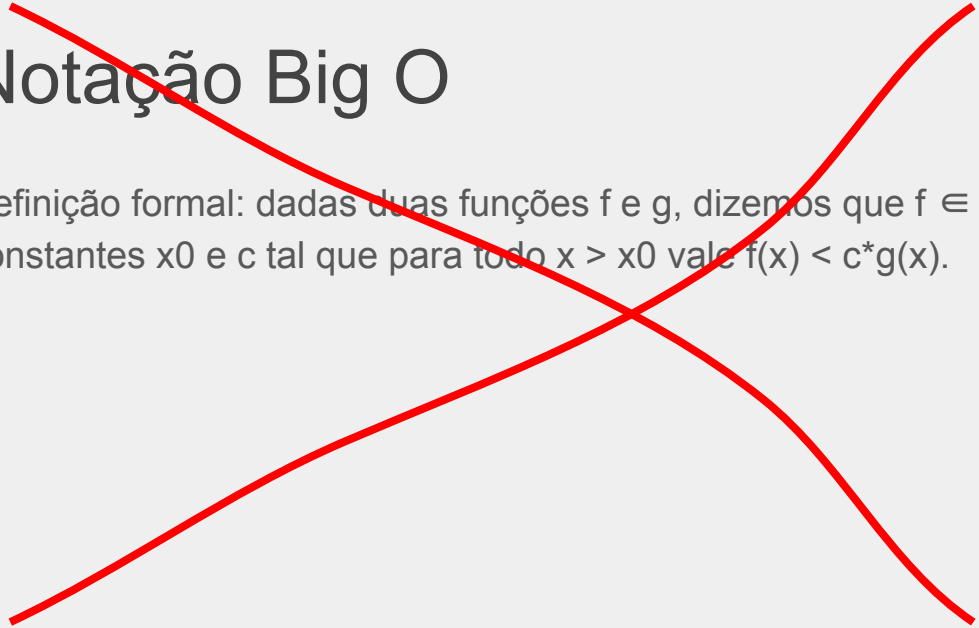
Notação Big O

Notação Big O

Definição formal: dadas duas funções f e g, dizemos que $f \in O(g)$ se existem constantes x0 e c tal que para todo $x > x0$ vale $f(x) < c*g(x)$.

Notação Big O

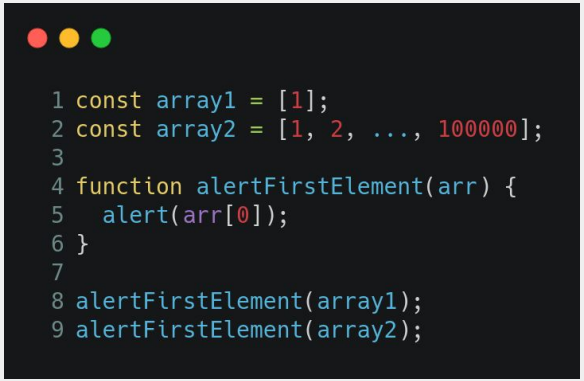
Definição formal: dadas duas funções f e g, dizemos que $f \in O(g)$ se existem constantes x0 e c tal que para todo $x > x0$ vale $f(x) < c*g(x)$.



Notação Big O

Definição informal (logo, menos precisa): Queremos saber o comportamento do algoritmo a medida que a entrada aumenta, isso implica em saber quais termos (trecho de código) que crescem mais rápidos, ou seja, os que dominam o valor total.

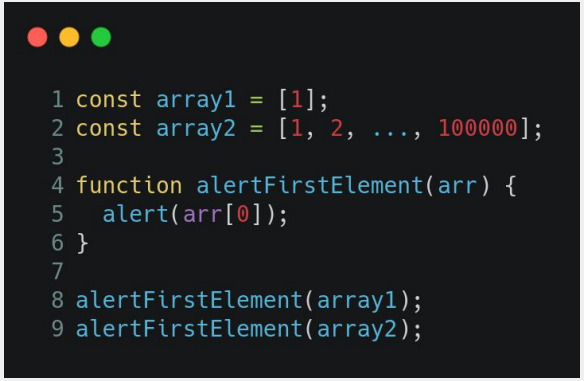
Notação Big O - Exemplo



```
1 const array1 = [1];
2 const array2 = [1, 2, ..., 100000];
3
4 function alertFirstElement(arr) {
5   alert(arr[0]);
6 }
7
8 alertFirstElement(array1);
9 alertFirstElement(array2);
```

Notação Big O - Exemplo

Como podemos classificar a quantidade de tempo necessária para essa função terminar de ser executada? As entradas de tamanhos diferentes terão tempos de execução diferentes?



```
1 const array1 = [1];
2 const array2 = [1, 2, ..., 100000];
3
4 function alertFirstElement(arr) {
5   alert(arr[0]);
6 }
7
8 alertFirstElement(array1);
9 alertFirstElement(array2);
```

Notação Big O - Exemplo

Como podemos classificar a quantidade de tempo necessária para essa função terminar de ser executada? As entradas de tamanhos diferentes terão tempos de execução diferentes?

```
1 const array1 = [1];
2 const array2 = [1, 2, ..., 100000];
3
4 function alertFirstElement(arr) {
5   alert(arr[0]);
6 }
7
8 alertFirstElement(array1);
9 alertFirstElement(array2);
```

E se substituirmos a função acima por esta?

```
1 const array1 = [1];
2 const array2 = [1, 2, ..., 100000];
3
4 function alertAllElements(arr) {
5   arr.forEach(elem => alert(elem));
6 }
7
8 alertAllElements(array1);
9 alertAllElements(array2);
```

Notação Big O - Exemplo

Independentemente de quantos elementos estão contidos na entrada dessa primeira função, ela sempre operará uma vez.
Uma operação → O(1)

```
const array1 = [1];
const array2 = [1, 2, ..., 100000];

function alertFirstElement(array) {
  alert(array[0]);
}

alertFirstElement(array1);
alertFirstElement(array2);
```

```
let array1 = [1];
let array2 = [1, 2, ..., 3000];

function alertAllElements(array) {
  array.forEach(element => alert(element));
};

alertFirstElement(array1);
alertFirstElement(array2);
```

Notação Big O - Exemplo

Independentemente de quantos elementos estão contidos na entrada dessa primeira função, ela sempre operará uma vez.
Uma operação → O(1)

```
const array1 = [1];
const array2 = [1, 2, ..., 100000];

function alertFirstElement(array) {
  alert(array[0]);
}

alertFirstElement(array1);
alertFirstElement(array2);
```

Esta no entanto, a quantidade de operações está diretamente relacionado ao tamanho da entrada que ela recebe**!**
n operações → O(*n*)

```
let array1 = [1];
let array2 = [1, 2, ..., 3000];

function alertAllElements(array) {
  array.forEach(element => alert(element));
};

alertFirstElement(array1);
alertFirstElement(array2);
```

Em suma: Big O Notation é usado para classificar algoritmos pela rapidez com que seus tempos de execução crescem em relação à sua entrada. Vide GIF:

O que não se pode medir, não se pode melhorar!!

Notação Big O
Os casos mais comuns

Notação Big O

- O(1) - Tempo constante
- O(log$n$) - Tempo logarítmico
- O(n) - Tempo linear
- O($n$log$n$) - Tempo linear~quadrático
- O(n^2) - Tempo quadrático

Notação Big O

- O(1) - Tempo constante
- O(log*n*) - Tempo logarítmico
- O(*n*) - Tempo linear
- O(*n*log*n*) - Tempo linear~quadrático
- O(n^2) - Tempo quadrático

Quanto mais acima estiver,
melhor é a complexidade!

Notação Big O

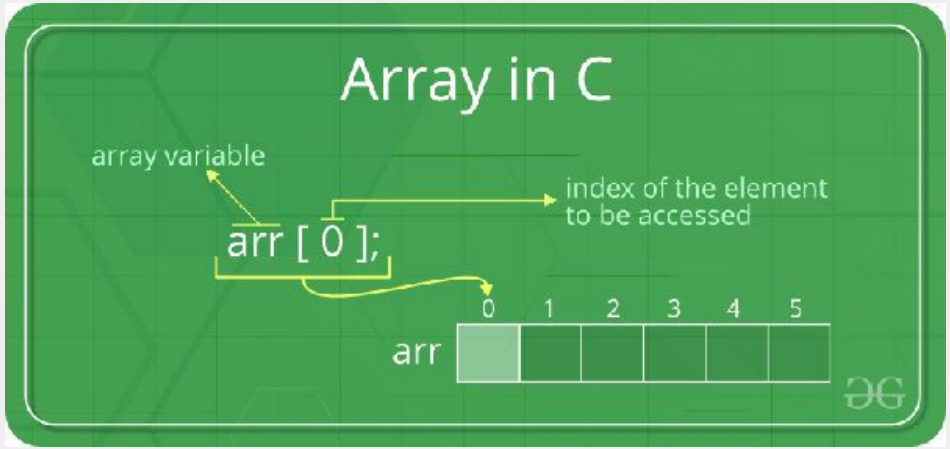
- O(1)
- O(log$n$)
- O(n)
- O($n$log$n$)
- O(n^2)

Notação Big O

- O(1)

  Exemplo 1: array em C

  Em C, para acessar uma posição diferente de 0, é calculado o offset e somado a posição 0. Tornando o acesso direto ou O(1)!



Notação Big O

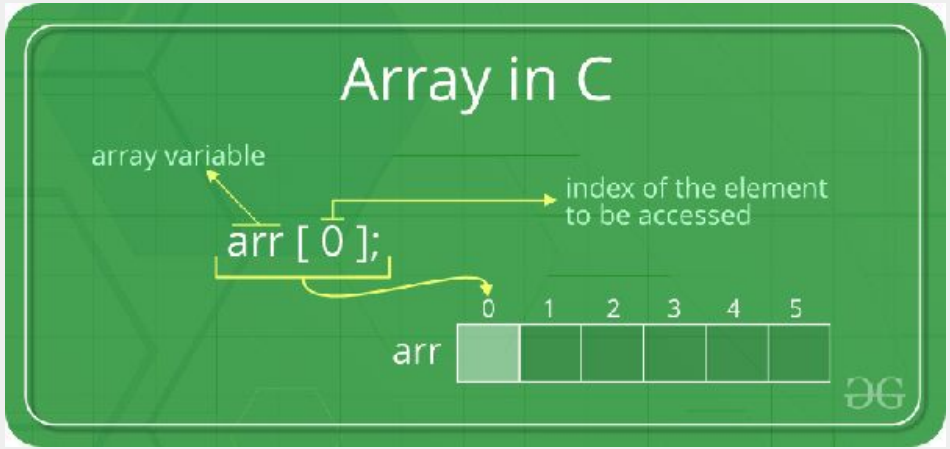
- O(1)

  Exemplo 1: array em C

  Em C, para acessar uma posição diferente de 0, é calculado o offset e somado a posição 0. Tornando o acesso direto!

Por exemplo:

arr[3] = *(arr[0] + offset)



Notação Big O

- O(1)

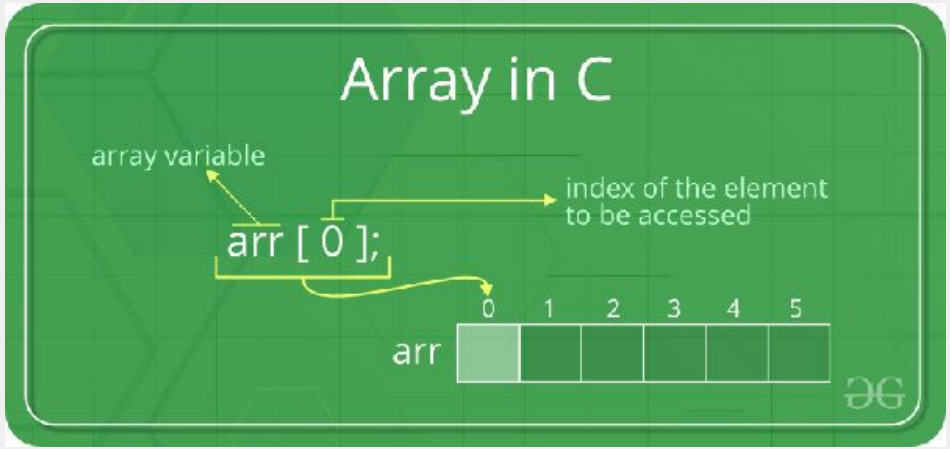
  Exemplo 1: array em C

  Em C, para acessar uma posição diferente de 0, é calculado o offset e somado a posição 0. Tornando o acesso direto!

Por exemplo:

arr[3] = *(arr[0] + offset)

Logo, se tentar acessar uma posição menor que 0 ou maior que 5 dará erro. Pois só há referência entre 0 e 5.

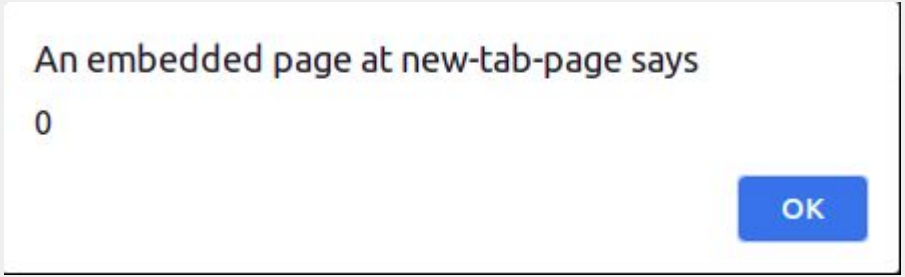


Notação Big O

- O(1)

  Exemplo 2: array em JavaScript

```
1 const array = []
2 array[2] = 2
3 array[1] = 1
4 array[0] = 0
5
6 alert(array[0])
```



Notação Big O

- O(1)

  Exemplo 2: array em JavaScript

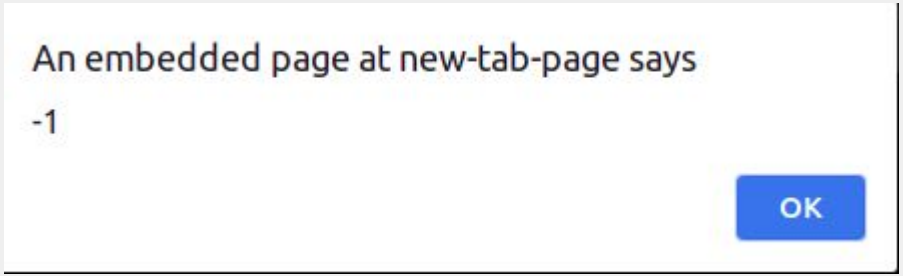
```
1 const array = []
2 array[2] = 2
3 array[1] = 1
4 array[0] = 0
5 array[-1] = -1
6
7 alert(array[-1])
```

Notação Big O

- O(1)

  Exemplo 2: array em JavaScript

```
1 const array = []
2 array[2] = 2
3 array[1] = 1
4 array[0] = 0
5 array[-1] = -1
6
7 alert(array[-1])
```



Notação Big O

- O(1)

  Exemplo 3: hash table/hash map/dictionary (ou em JavaScript: object)

Notação Big O

- O(1)
- O(log*n*)
- O(*n*)
- O(*n*log*n*)
- O(n^2)

Notação Big O

- O(1)
- O(log*n*)

  Exemplo 1: busca binária

Notação Big O

- O(1)
- O(log*n*)

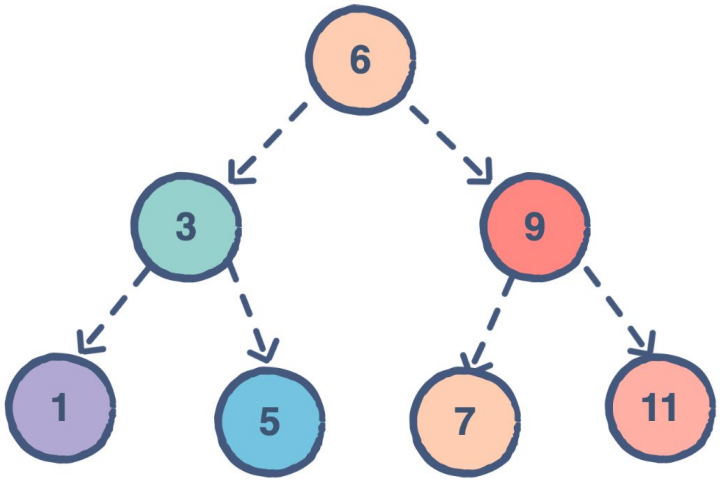
  Exemplo 1: busca binária

A cada passo reduzimos a busca a metade. Daí que vem a relação logarítmica! Pois no pior caso, levaremos log(n) passos. ou, no nosso caso: log(8) = 3 passos.

Notação Big O

- O(1)
- O(log*n*)

Exemplo 2: search em uma árvore binária de busca balanceada



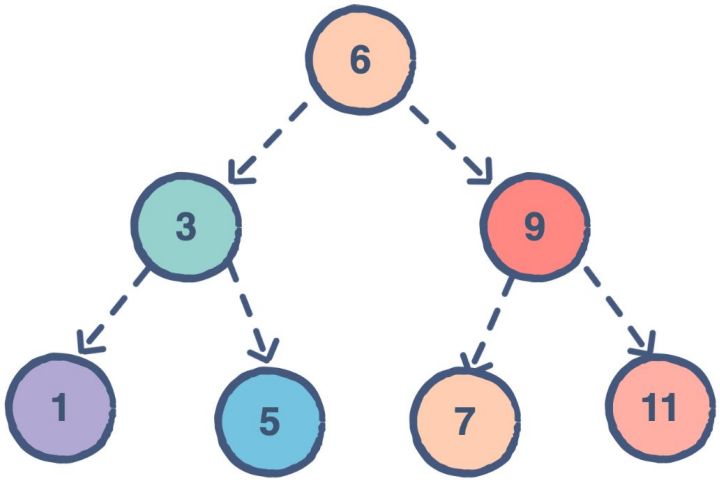
An example of a binary search tree

Notação Big O

- O(1)
- O(log*n*)

Exemplo 2: search em uma árvore binária de busca balanceada

No pior caso, a quantidade de passos será o tamanho da árvore. Sua altura é dada por: altura = log(*n*), onde *n* é a quantidade de nós.



An example of a binary search tree

Notação Big O

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$

Notação Big O

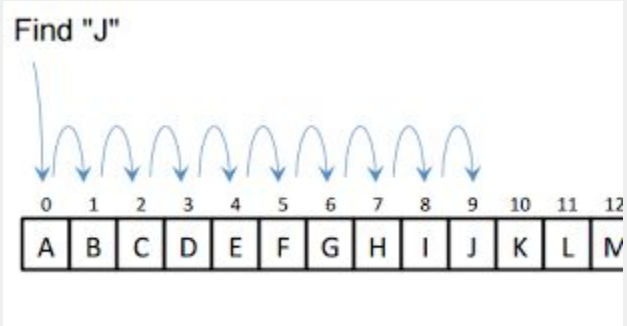
- O(1)
- O(log*n*)
- O(*n*)

Já vimos lá atrás…

Notação Big O

- O(1)
- O(log*n*)
- O(*n*)

  Exemplo 1: busca sequencial.

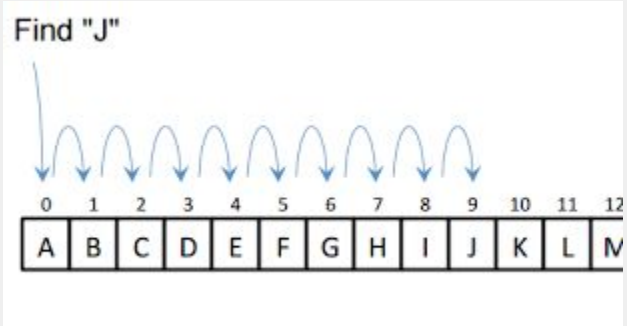


Notação Big O

- O(1)
- O(log*n*)
- O(*n*)

Exemplo 1: busca sequencial.

Busca um a um! No pior caso precisamos percorrer os n elementos do array. Logo: O(*n*).



Notação Big O

- O(1)
- O(log*n*)
- O(*n*)

Exemplo 2: alertar um array (percorrer um array).

Um único loop!

```
1 function alertArray(arr) {
2   for (let indice = 0; indice < arr.length; indice++)
3     alert(arr[indice])
4 }
```

Notação Big O

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$

Notação Big O

- O(1)
- O(log*n*)
- O(*n*)
- O(*n*log*n*)

Exemplo: Sort - Geralmente sorts são implementados em O(*n*log*n*).

Notação Big O

- O(1)
- O(log*n*)
- O(*n*)
- O(*n*log*n*)

Exemplo: Sort.

Array.prototype.sort()

A complexidade do tempo de execução ou a quantidade de memória utilizada pela ordenação não pode ser garantido e depende da implementação realizada.

Sintaxe

```
arr.sort([funcaoDeComparacao])
```

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

Notação Big O

- O(1)
- O(log*n*)
- O(*n*)
- O(*n*log*n*)
- O(n^2)

Notação Big O

- O(1)
- O(log*n*)
- O(*n*)
- O(*n*log*n*)
- O(n^2)

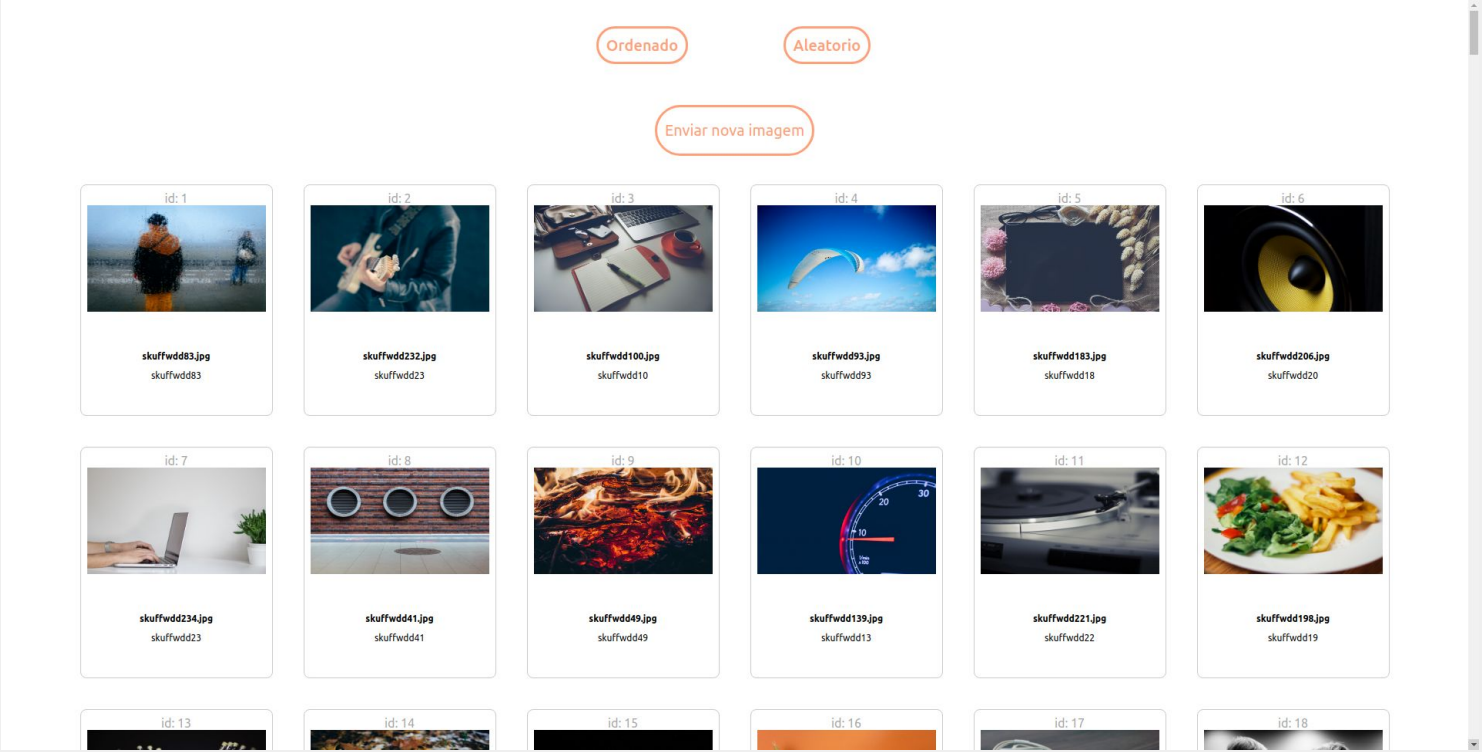
Exemplo: alertar uma matriz (percorrer uma matriz).

```
1 function alertMatrix(matrix) {
2   for (let row = 0; row < matrix.length; row++)
3     for (let column = 0; column < matrix[0].length; column++)
4       alert(matrix[row][column])
5 }
```

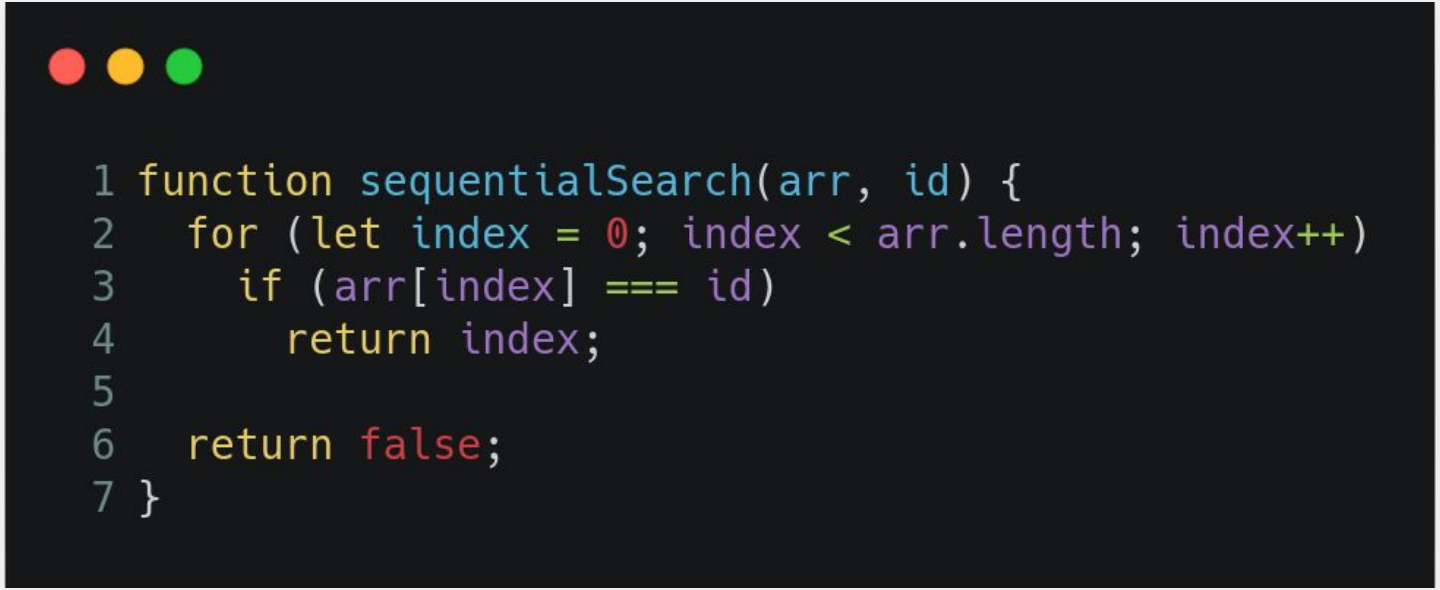
Recapitulando o exemplo...

Use case: o usuário deseja poder pesquisar o card pelo id.

*É garantido que os cards estão ordenados pelo id. Por baixo dos panos, o que o front precisa fazer é retornar o índice do id no array (se existir).



Busca sequencial - Complexidade: O(*n*)



```
1 function sequentialSearch(arr, id) {
2   for (let index = 0; index < arr.length; index++)
3     if (arr[index] === id)
4       return index;
5
6   return false;
7 }
```

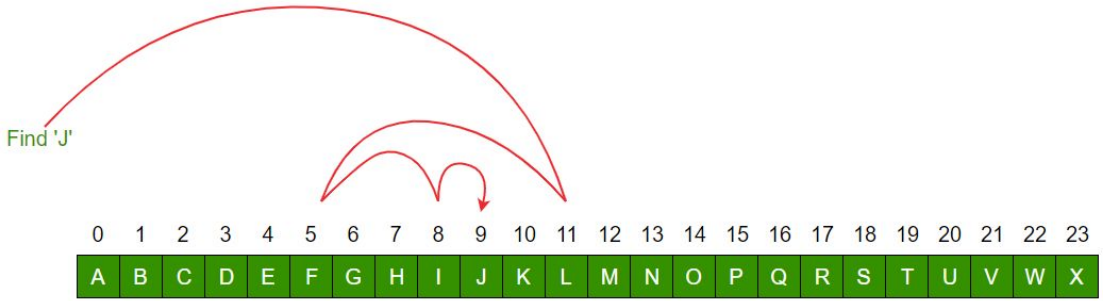
Busca sequencial - Complexidade: O(*n*)

Apenas 1 loop!

```
1 function sequentialSearch(arr, id) {
2   for (let index = 0; index < arr.length; index++)
3     if (arr[index] === id)
4       return index;
5
6   return false;
7 }
```

Com base no que a gente já viu até agora, alguma ideia de como resolver esse problema mais rápido?

Busca binária!



Busca binária - O(log*n*)

```
function binarySearch(arr, x, start, end) {
  if (start > end) return false;

  const mid = Math.floor((start + end) / 2);

  if (arr[mid] === x) return mid;

  if (arr[mid] > x) return binarySearch(arr, x, start, mid - 1);

  return binarySearch(arr, x, mid + 1, end);
}
```

Pensou no método find? (Array.prototype.find(x))

Pensou no método find? (Array.prototype.find(x))
Qual seria a complexidade nesse caso?

Pensou no método find? (Array.prototype.find(x))

Qual seria a complexidade nesse caso?

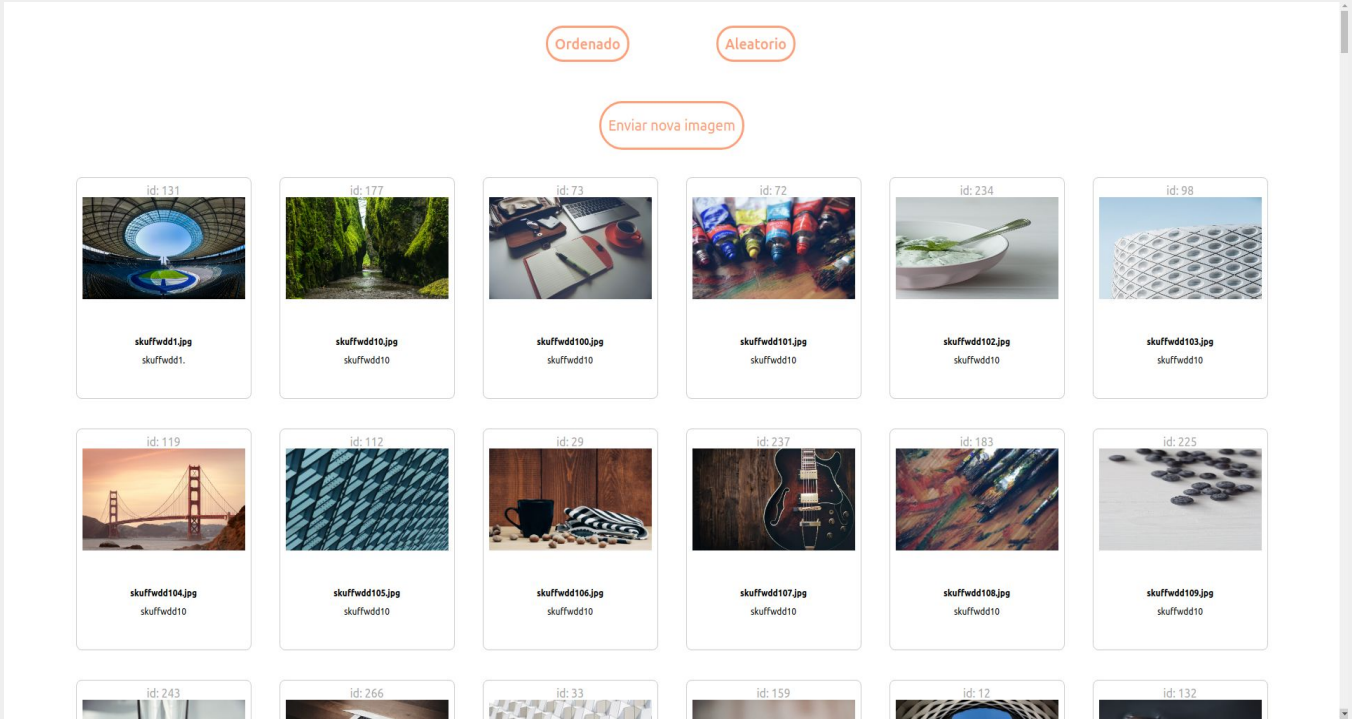
O(n)!!

Pensou no método find? (Array.prototype.find(x))
Qual seria a complexidade nesse caso?
O(n)!!
É sempre bom ter cuidado nas soluções rápidas de serem codadas...

Case 2

Use case: o usuário poderá fazer o upload de uma nova imagem.

*Dado que os cards NÃO estão ordenados pelo id, precisa-se descobrir qual o menor id que não está listado nos cards.



Exemplo

Dado o array de ids: [6, 2, 1, 7];

O menor id que não está listado é o 3.

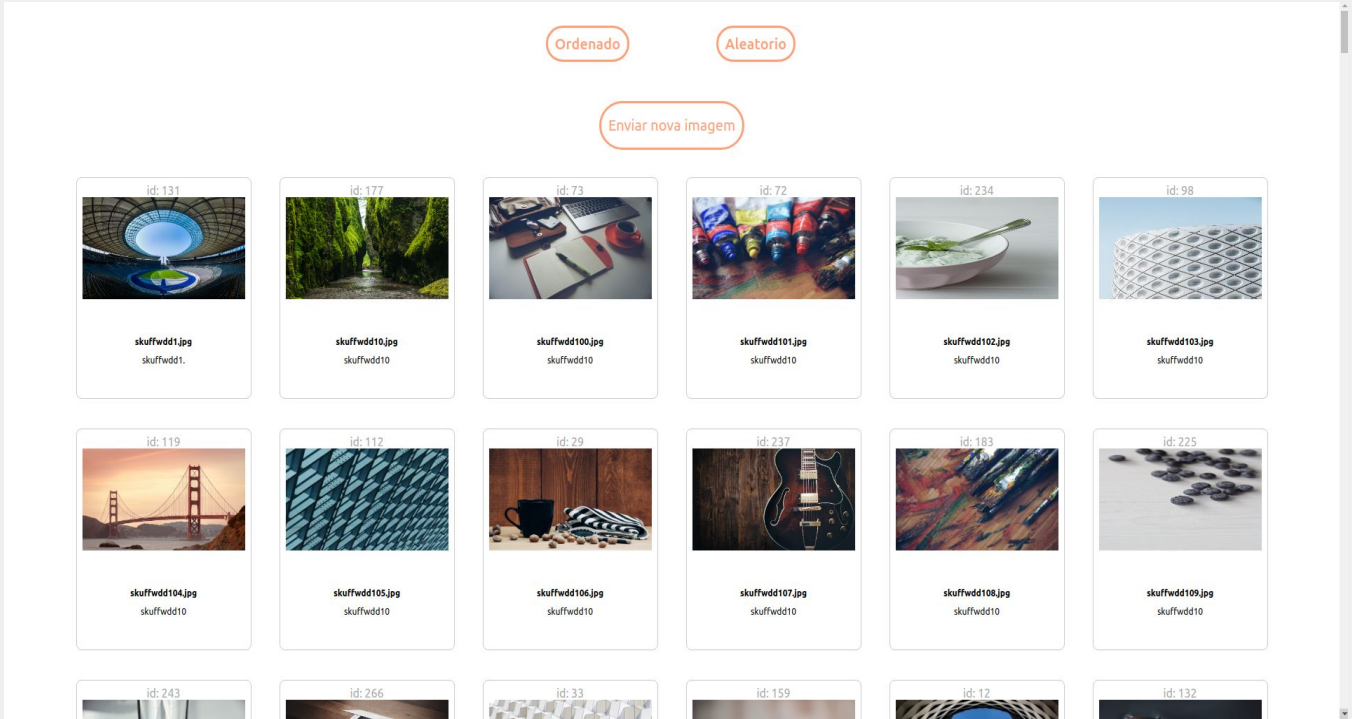
Outro exemplo

Dado o array de ids: [4, 2, 1, 3];

O menor id que não está listado é o 5 (o tamanho do vetor + 1).

Use case: o usuário poderá fazer o upload de uma nova imagem.

*Dado que os cards NÃO estão ordenados pelo id, precisa-se descobrir qual o menor id que não está listado nos cards.



Ideias?

Gerando ideias - Brute force

Array: [5,2,1,9].

- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.

Gerando ideias - Brute force

Array: [5,2,1,9].

- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.
- Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante até o tamanho do array...

Gerando ideias - Brute force

Array: [5,2,1,9].
Elegíveis para menor id: [1,2,3,4].

- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.
- Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante até o tamanho do array...

Gerando ideias - Brute force

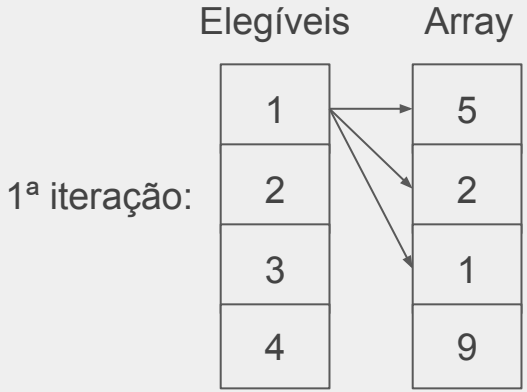
Array: [5,2,1,9].
Elegíveis para menor id: [1,2,3,4].

- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.
- Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante até o tamanho do array...
- Em código, podemos fazer um loop variando de 1 até o tamanho do array (*n*), e para cada iteração é pesquisado no array se o id está nele.

Gerando ideias - Brute force

Array: [5,2,1,9].
Elegíveis para menor id: [1,2,3,4].

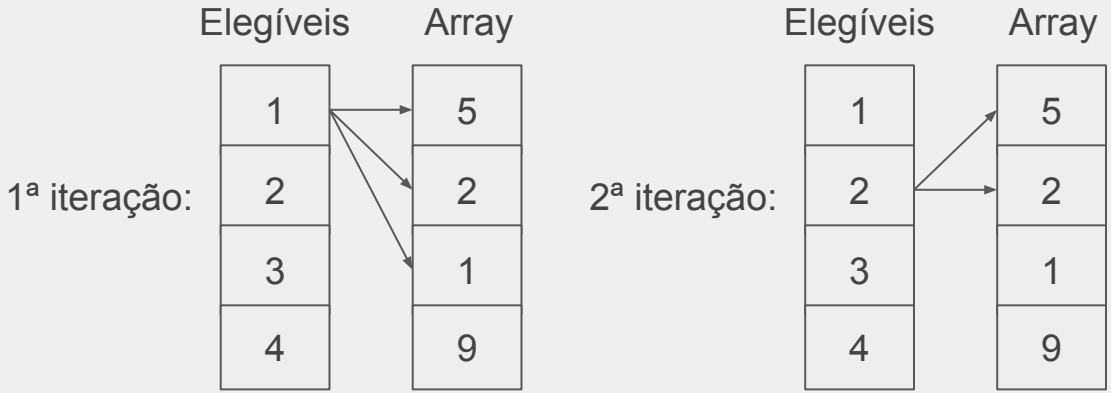
- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.
- Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante até o tamanho do array...
- Em código, podemos fazer um loop variando de 1 até o tamanho do array (*n*), e para cada iteração é pesquisado no array se o id está nele.



Gerando ideias - Brute force

Array: [5,2,1,9].
Elegíveis para menor id: [1,2,3,4].

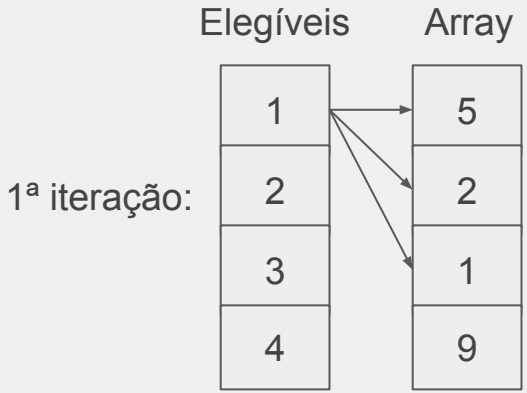
- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.
- Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante até o tamanho do array...
- Em código, podemos fazer um loop variando de 1 até o tamanho do array (*n*), e para cada iteração é pesquisado no array se o id está nele.

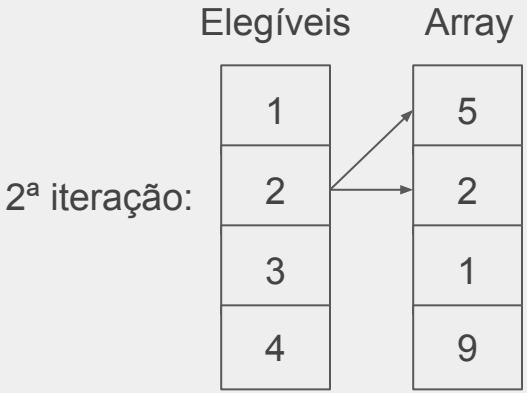


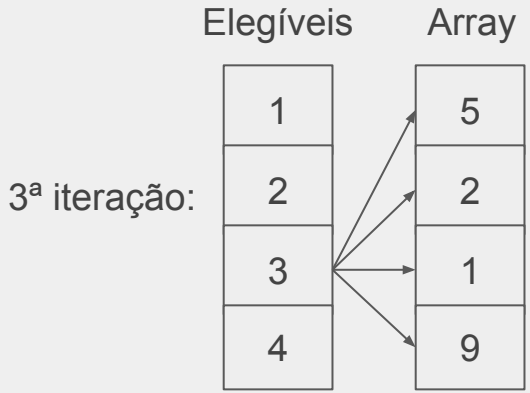
Gerando ideias - Brute force

Array: [5,2,1,9].
Elegíveis para menor id: [1,2,3,4].

- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.
- Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante até o tamanho do array...
- Em código, podemos fazer um loop variando de 1 até o tamanho do array (*n*), e para cada iteração é pesquisado no array se o id está nele.



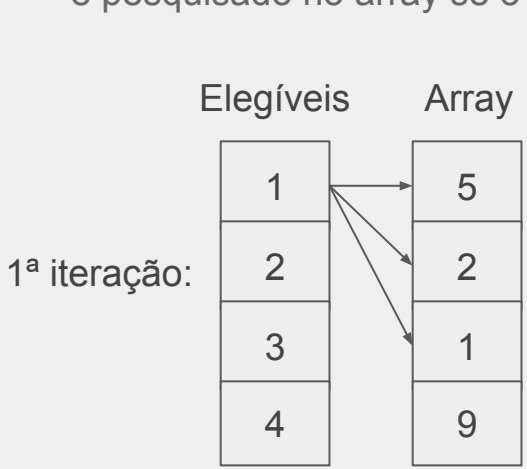


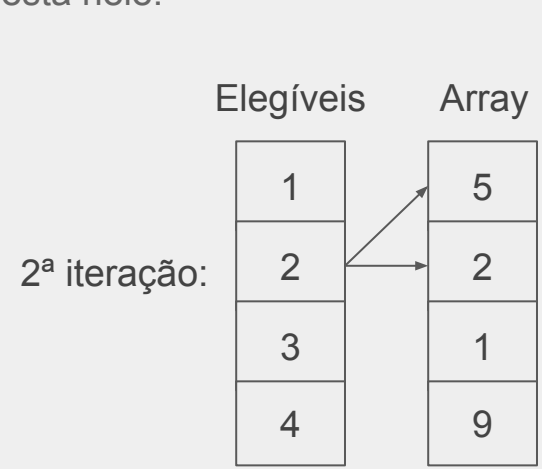


Gerando ideias - Brute force

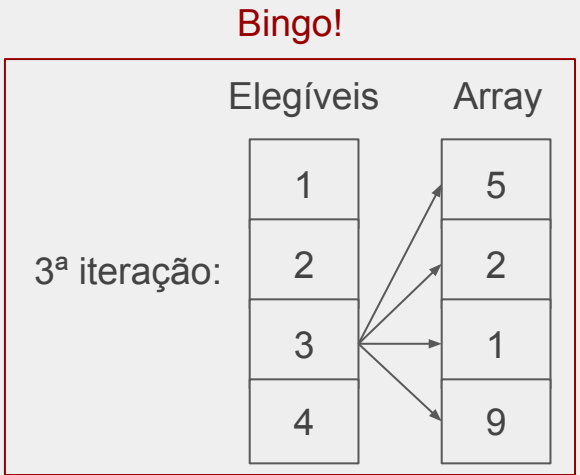
Array: [5,2,1,9].
Elegíveis para menor id: [1,2,3,4].

- O menor id possível é 1. O maior id possível é o tamanho do array mais 1. No caso, 5.
- Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante até o tamanho do array...
- Em código, podemos fazer um loop variando de 1 até o tamanho do array (*n*), e para cada iteração é pesquisado no array se o id está nele.

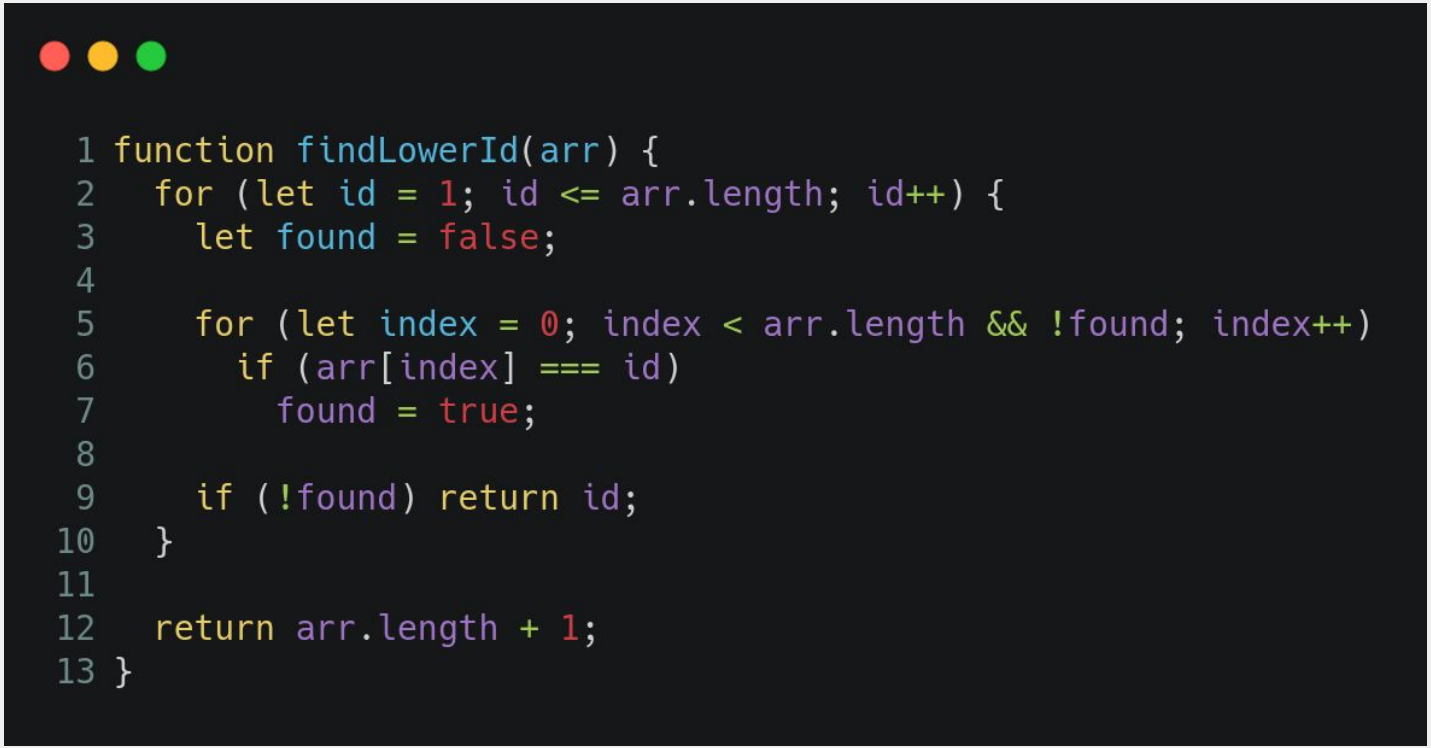




Bingo!



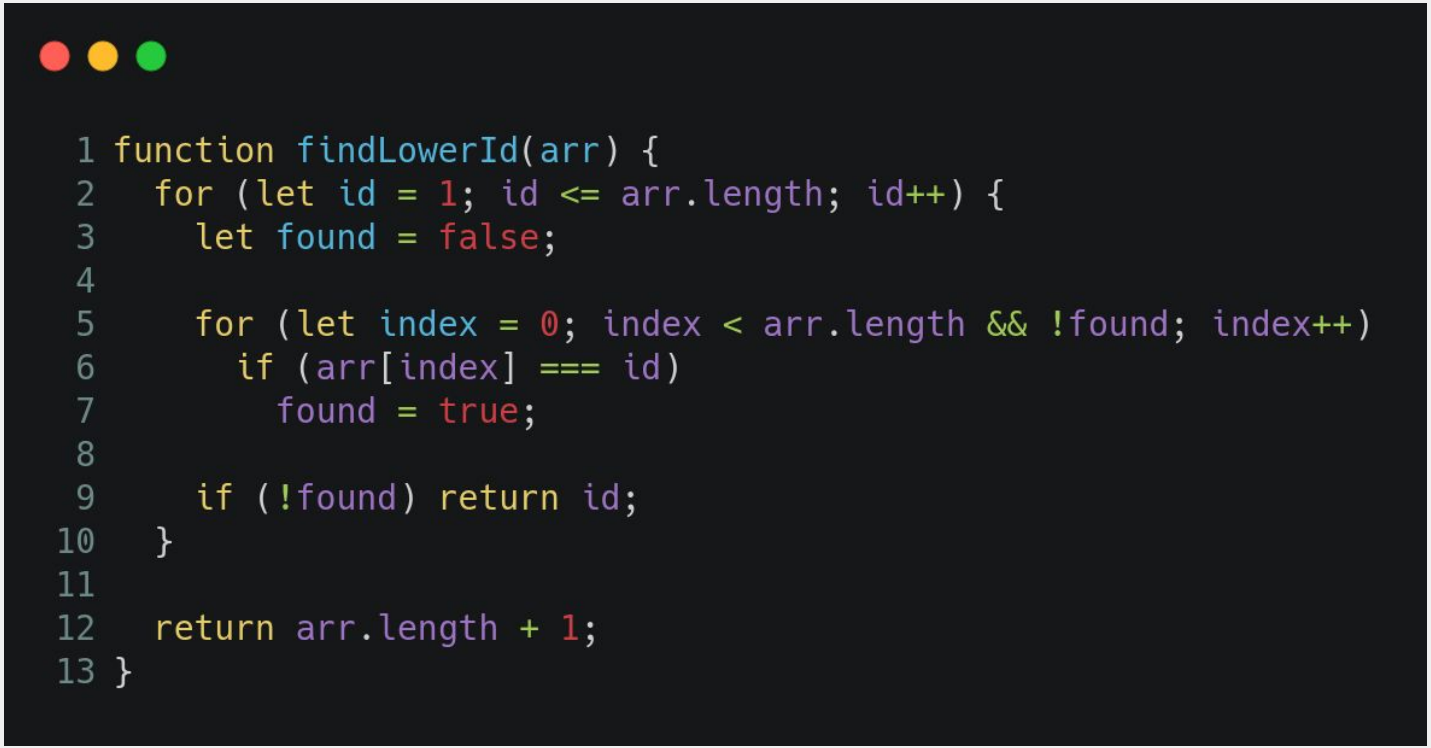
Solução - Brute force



```
1 function findLowerId(arr) {
2   for (let id = 1; id <= arr.length; id++) {
3     let found = false;
4
5     for (let index = 0; index < arr.length && !found; index++)
6       if (arr[index] === id)
7         found = true;
8
9     if (!found) return id;
10  }
11
12  return arr.length + 1;
13 }
```

Solução - Brute force

Qual a complexidade?



```
function findLowerId(arr) {
  for (let id = 1; id <= arr.length; id++) {
    let found = false;

    for (let index = 0; index < arr.length && !found; index++)
      if (arr[index] === id)
        found = true;

    if (!found) return id;
  }

  return arr.length + 1;
}
```

Solução - Brute force

Qual a complexidade?

$O(n^2)$!

Pois para cada id testado (loop externo), percorremos todo o array (loop interno).

```
 1 function findLowerId(arr) {
 2   for (let id = 1; id <= arr.length; id++) {
 3     let found = false;
 4
 5     for (let index = 0; index < arr.length && !found; index++)
 6       if (arr[index] === id)
 7         found = true;
 8
 9     if (!found) return id;
10   }
11
12   return arr.length + 1;
13 }
```

Alguma forma mais rápida?

Gerando ideias - Sort

Se ordenarmos o vetor, podemos fazer uma busca sequencial!

Gerando ideias - Sort

```
1 function findLowerId(arr) {
2   let sortedArr = arr.sort();
3
4   for (let index = 0; index < arr.length; index++) {
5     const id = index + 1;
6     if (sortedArr[index] != id) return id;
7   }
8
9   return arr.length + 1;
10 }
```

Gerando ideias - Sort

Qual a complexidade?

```
1 function findLowerId(arr) {
2   let sortedArr = arr.sort();
3
4   for (let index = 0; index < arr.length; index++) {
5     const id = index + 1;
6     if (sortedArr[index] != id) return id;
7   }
8
9   return arr.length + 1;
10 }
```

Gerando ideias - Sort

Qual a complexidade?

O(*n*) ? 😄

```
1  function findLowerId(arr) {
2    let sortedArr = arr.sort();
3
4    for (let index = 0; index < arr.length; index++) {
5      const id = index + 1;
6      if (sortedArr[index] != id) return id;
7    }
8
9    return arr.length + 1;
10 }
```

Gerando ideias - Sort

Ops... Não podemos esquecer do sort!

```
1 function findLowerId(arr) {
2   let sortedArr = arr.sort();
3
4   for (let index = 0; index < arr.length; index++) {
5     const id = index + 1;
6     if (sortedArr[index] != id) return id;
7   }
8
9   return arr.length + 1;
10 }
```

Qual a complexidade?

Gerando ideias - Sort

Qual a complexidade?

O(*n*log*n* + *n*)

```
1 function findLowerId(arr) {
2   let sortedArr = arr.sort();
3
4   for (let index = 0; index < arr.length; index++) {
5     const id = index + 1;
6     if (sortedArr[index] != id) return id;
7   }
8
9   return arr.length + 1;
10 }
```

Gerando ideias - Sort

Predominante!!

```
1  function findLowerId(arr) {
2    let sortedArr = arr.sort();
3
4    for (let index = 0; index < arr.length; index++) {
5      const id = index + 1;
6      if (sortedArr[index] != id) return id;
7    }
8
9    return arr.length + 1;
10 }
```

Qual a complexidade?

O(*n*log*n*)

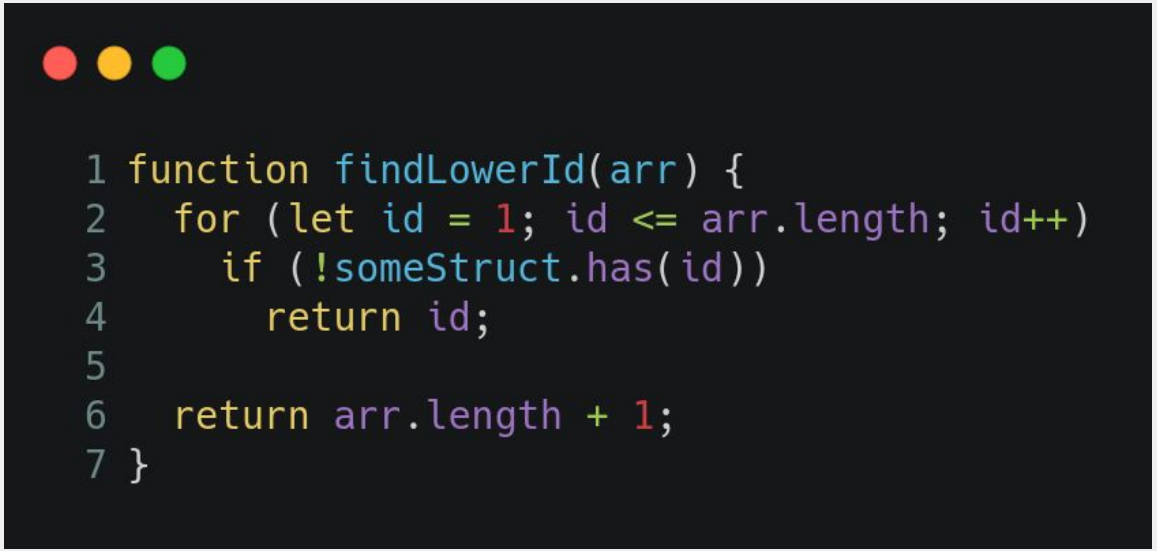
Massa!! Conseguimos diminuir de $O(n^2)$ para $O(n \log n)$

Alguma forma mais otimizada?

Gerando ideias

Vimos que o gargalo é no sort… Há uma forma de saber se o id existe em O(1)?

Algo como:



```
function findLowerId(arr) {
  for (let id = 1; id <= arr.length; id++)
    if (!someStruct.has(id))
      return id;

  return arr.length + 1;
}
```

Gerando ideias

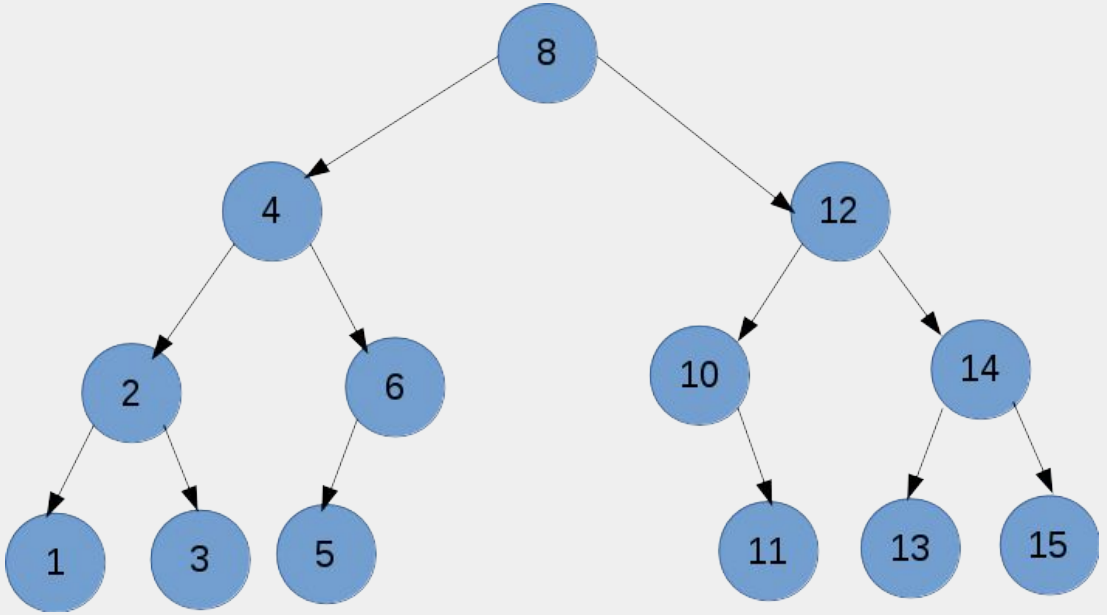
Vimos que o gargalo é no sort… Há uma forma de saber se o id existe em O(1)?

Algo como:

Seria perfeito pois a complexidade seria O(*n*)!!

```
1 function findLowerId(arr) {
2   for (let id = 1; id <= arr.length; id++)
3     if (!someStruct.has(id))
4       return id;
5
6   return arr.length + 1;
7 }
```

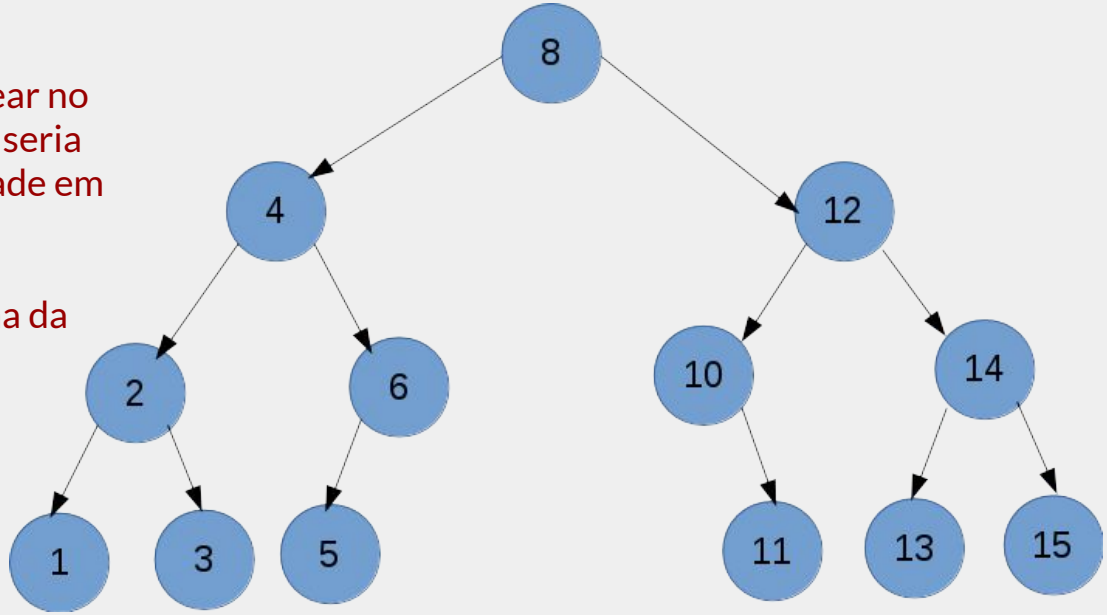
Gerando ideias - Árvore binária



Gerando ideias - Árvore binária

Parece bom... Mas além de balancear no processo de criação (AVL), a busca seria em O(log*n*), deixando a complexidade em O(*n*log*n*).

Então, a complexidade fica a mesma da solução com Sort!

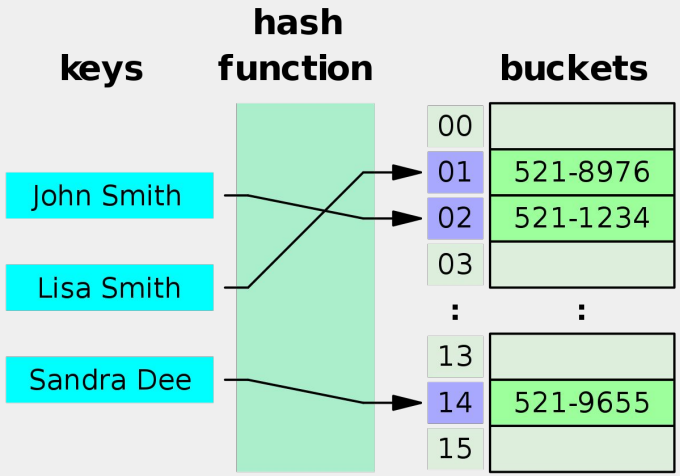


Queremos saber se um determinado id existe em O(1)!!!

Hash table

Hash table

É uma estrutura de dados que implementa um tipo de dados abstratos, pode mapear chaves para valores. Uma tabela de hash usa uma função de hash para calcular um índice, também chamado de código de hash, em uma array de buckets ou slots, a partir dos quais o valor desejado pode ser encontrado no caso médio em O(1).



Solução - Hash table

```
 1 function findLowerIdHashTable(arr) {
 2   const hashTable = new Map( );
 3
 4   for (let index = 0; index < arr.length; index++)
 5     hashTable.set(arr[index]);
 6
 7   for (let id = 1; id <= arr.length; id++)
 8     if (!hashTable.has(id)) return id;
 9
10   return arr.length + 1;
11 }
```

Solução - Hash table

Qual a complexidade?

```
1 function findLowerIdHashTable(arr) {
2   const hashTable = new Map();
3
4   for (let index = 0; index < arr.length; index++)
5     hashTable.set(arr[index]);
6
7   for (let id = 1; id <= arr.length; id++)
8     if (!hashTable.has(id)) return id;
9
10  return arr.length + 1;
11 }
```

Solução - Hash table

Qual a complexidade?

$O(n + n) = O(2n)$

```
1 function findLowerIdHashTable(arr) {
2   const hashTable = new Map();
3
4   for (let index = 0; index < arr.length; index++)
5     hashTable.set(arr[index]);
6
7   for (let id = 1; id <= arr.length; id++)
8     if (!hashTable.has(id)) return id;
9
10  return arr.length + 1;
11 }
```

Solução - Hash table

Qual a complexidade?

O(*n*)

```
 1 function findLowerIdHashTable(arr) {
 2   const hashTable = new Map();
 3
 4   for (let index = 0; index < arr.length; index++)
 5     hashTable.set(arr[index]);
 6
 7   for (let id = 1; id <= arr.length; id++)
 8     if (!hashTable.has(id)) return id;
 9
10   return arr.length + 1;
11 }
```

Evolução das soluções

1. Brute Force → $O(n^2)$
2. Sort/Tree → $O(n \log n)$
3. Hash table → $O(n)$

Hash table x Brute force

```
1 function findLowerIdHashTable(arr) {
2   const hashTable = new Map();
3
4   for (let index = 0; index < arr.length; index++)
5     hashTable.set(arr[index]);
6
7   for (let id = 1; id <= arr.length; id++)
8     if (!hashTable.has(id)) return id;
9
10  return arr.length + 1;
11 }
```

O(2*n*)

O(*n*²)

```
1 function findLowerIdBruteForce(arr) {
2   for (let id = 1; id <= arr.length; id++) {
3     let found = false;
4
5     for (let index = 0; index < arr.length && !found; index++)
6       if (arr[index] === id)
7         found = true;
8
9     if (!found) return id;
10  }
11
12  return arr.length + 1;
13 }
```

Hash table x Brute force

Pegando um exemplo supracitado:

```
const array = [1, 2, ..., 100000]
```

```
function findLowerIdHashTable(arr) {
  const hashTable = new Map();

  for (let index = 0; index < arr.length; index++)
    hashTable.set(arr[index]);

  for (let id = 1; id <= arr.length; id++)
    if (!hashTable.has(id)) return id;

  return arr.length + 1;
}
```

O(2n)

```
function findLowerIdBruteForce(arr) {
  for (let id = 1; id <= arr.length; id++) {
    let found = false;

    for (let index = 0; index < arr.length && !found; index++)
      if (arr[index] === id)
        found = true;

    if (!found) return id;
  }

  return arr.length + 1;
}
```

O(n^2)

Hash table x Brute force

```
const array = [1, 2, ..., 100000]
```

200 mil operações

```
1 function findLowerIdHashTable(arr) {
2   const hashTable = new Map();
3
4   for (let index = 0; index < arr.length; index++)
5     hashTable.set(arr[index]);
6
7   for (let id = 1; id <= arr.length; id++)
8     if (!hashTable.has(id)) return id;
9
10  return arr.length + 1;
11 }
```

O(2*n*)

O(n^2)

```
1 function findLowerIdBruteForce(arr) {
2   for (let id = 1; id <= arr.length; id++) {
3     let found = false;
4
5     for (let index = 0; index < arr.length && !found; index++)
6       if (arr[index] === id)
7         found = true;
8
9     if (!found) return id;
10  }
11
12  return arr.length + 1;
13 }
```

Hash table x Brute force

```
const array = [1, 2, ..., 100000]
```

200 mil operações

```
1 function findLowerIdHashTable(arr) {
2   const hashTable = new Map();
3
4   for (let index = 0; index < arr.length; index++)
5     hashTable.set(arr[index]);
6
7   for (let id = 1; id <= arr.length; id++)
8     if (!hashTable.has(id)) return id;
9
10  return arr.length + 1;
11 }
```

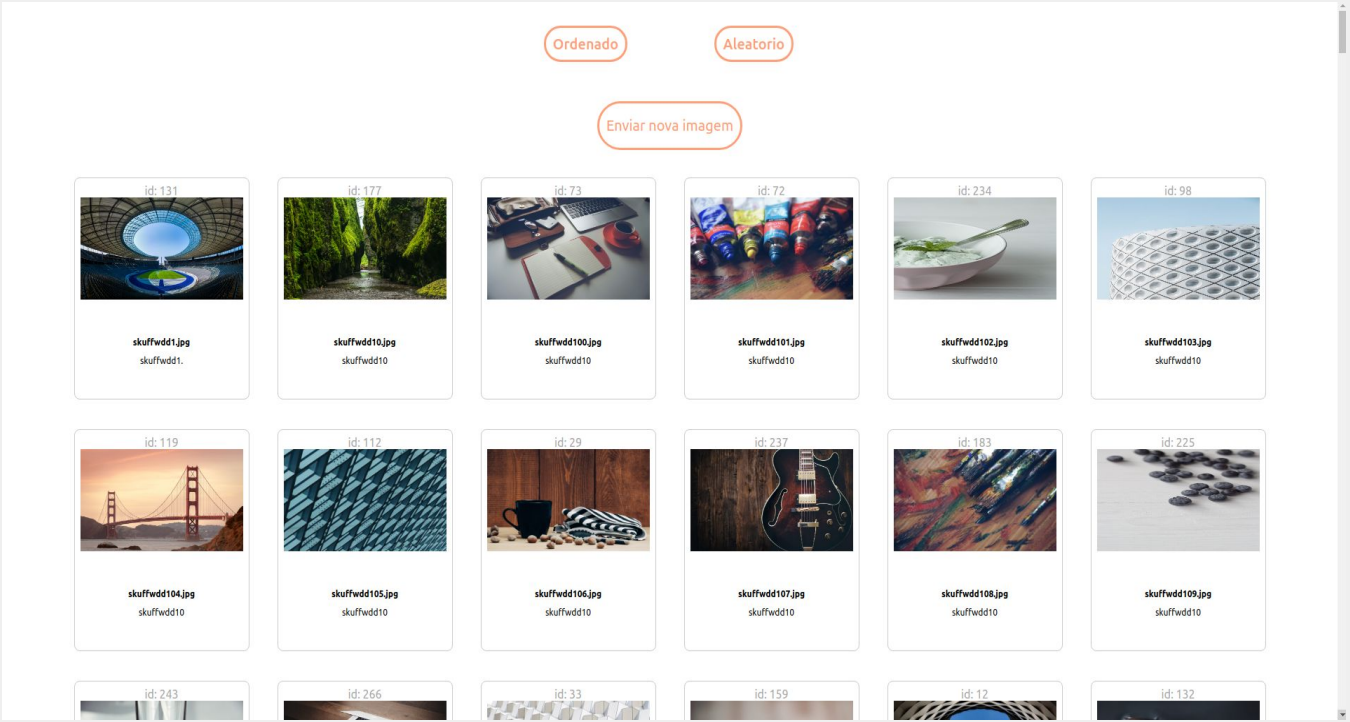
O(2*n*)

O(n^2)

10 bilhões de operações

```
1 function findLowerIdBruteForce(arr) {
2   for (let id = 1; id <= arr.length; id++) {
3     let found = false;
4
5     for (let index = 0; index < arr.length && !found; index++)
6       if (arr[index] === id)
7         found = true;
8
9     if (!found) return id;
10  }
11
12  return arr.length + 1;
13 }
```

Bônus - Use case: como encontrar o maior id?



Bônus - Use case: como encontrar o maior id?

Primeiro faz um Sort e depois pega o último elemento?

Bônus - Use case: como encontrar o maior id?

Primeiro faz um Sort e depois pega o último elemento?

Bônus - Use case: como encontrar o maior id?

Primeiro faz um Sort e depois pega o último elemento?

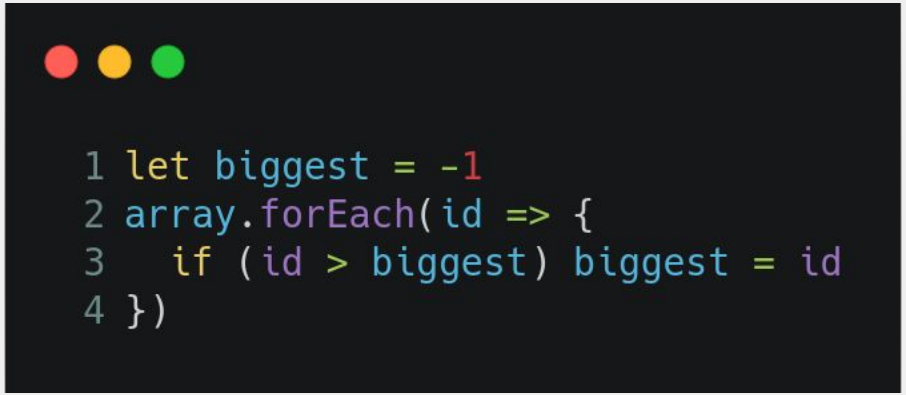
Como vimos, a complexidade do sort fica: O(*n*log*n*)

Bônus - Use case: como encontrar o maior id?

Primeiro faz um Sort e depois pega o último elemento?

Como vimos, a complexidade do sort fica: O(*n*log*n*)

Por que não um simples loop, criando uma variável auxiliar para guardar o maior valor enquanto percorre o array, deixando a complexidade em O(*n*)?



```
1 let biggest = -1
2 array.forEach(id => {
3   if (id > biggest) biggest = id
4 })
```

Bônus - Use case: como encontrar o maior id?

Primeiro faz um Sort e depois pega o último elemento?

Como vimos, a complexidade do sort fica: O(*n*log*n*)

Por que não um simples loop, criando uma variável auxiliar para guardar o maior valor enquanto percorre o array, deixando a complexidade em O(*n*)? (:

Quando temos conhecimento de vários algoritmos e estruturas de dados, as vezes queremos complicar o que é simples. Ou as vezes queremos codar rapidamente e nem sempre escolhemos as melhores soluções. Sempre é bom pensar um pouco mais...

Falando em pensar um pouco mais…

Extra talk, para pensar um pouco mais…

Use case: como encontrar os *k* maiores ids?

Extra talk, para pensar um pouco mais…

Use case: como encontrar os *k* menores ids?
Dica: a complexidade fica em O(*n*log*n*), O(nlogk) ou até mesmo O(n).

"Quando um dev quiser dar um passo a mais na carreira, naturalmente ele precisará estudar algoritmos e estruturas de dados"

Paulo Silveira, CEO Grupo Caelum Alura
https://hipsters.tech/algoritmos-e-estrutura-de-dados-hipsters-186

"Quando o conceito de excelência for intrínseco ao dev, naturalmente ele precisará estudar algoritmos e estruturas de dados"

Clarice Lispector + Gustavo Oliveira (não é nenhum CEO mas é o redator do slide!)
https://fakedofake.com

gustavooliveiraf

Fim!

Obrigado!

Gustavo Oliveira
Engenheiro de Software na Gympass



gustavooliveiraf

Dúvidas?