Otimizando o front-end: uma breve introdução sobre complexidade de tempo e estruturas de dados

Gustavo Oliveira

Antes de tudo, alguns avisos/esclarecimentos:

1. A apresentação terá muito código, mas o objetivo é entender a ideia;

2. As vezes vou fazer alguma pergunta, pausar a apresentação e interagir no chat. Enquanto isso, dá tempo de todo mundo assimilar o slide atual.

3. Entendendo a ideia, depois pode pesquisar se precisar. O problema é se prender ao código e desfocar de entender a ideia.

4. Será natural surgir dúvidas. Inclusive, para chegar a ter dúvidas foi preciso primeiro sair da inércia, isso pode ser um bom sinal (:

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

Alguns dados

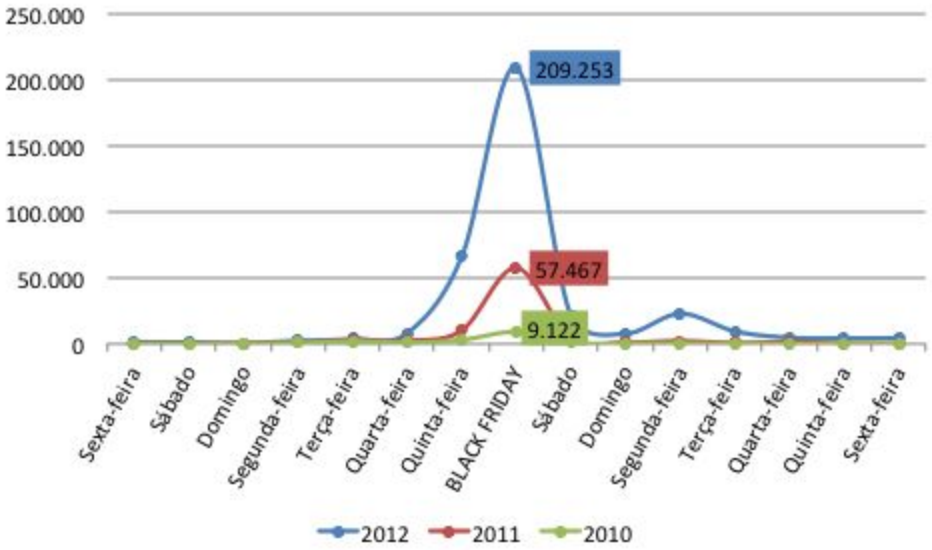
- 64% dos usuários tendem a deixar a página e buscar outro site que seja mais rápido;

- 79% dos usuários mais frequentes não compram em sites que são muito lentos.

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

Podemos ter degradação na performance em todas as áreas

Problema na infraestrutura!



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

Problemas no back-end?

Problemas no front-end?

Problemas no front-end?
Problemas na homepage?

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/

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

A otimização nesse ponto pode vir de várias áreas do conhecimento. Aqui vamos conhecer algumas...

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

"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

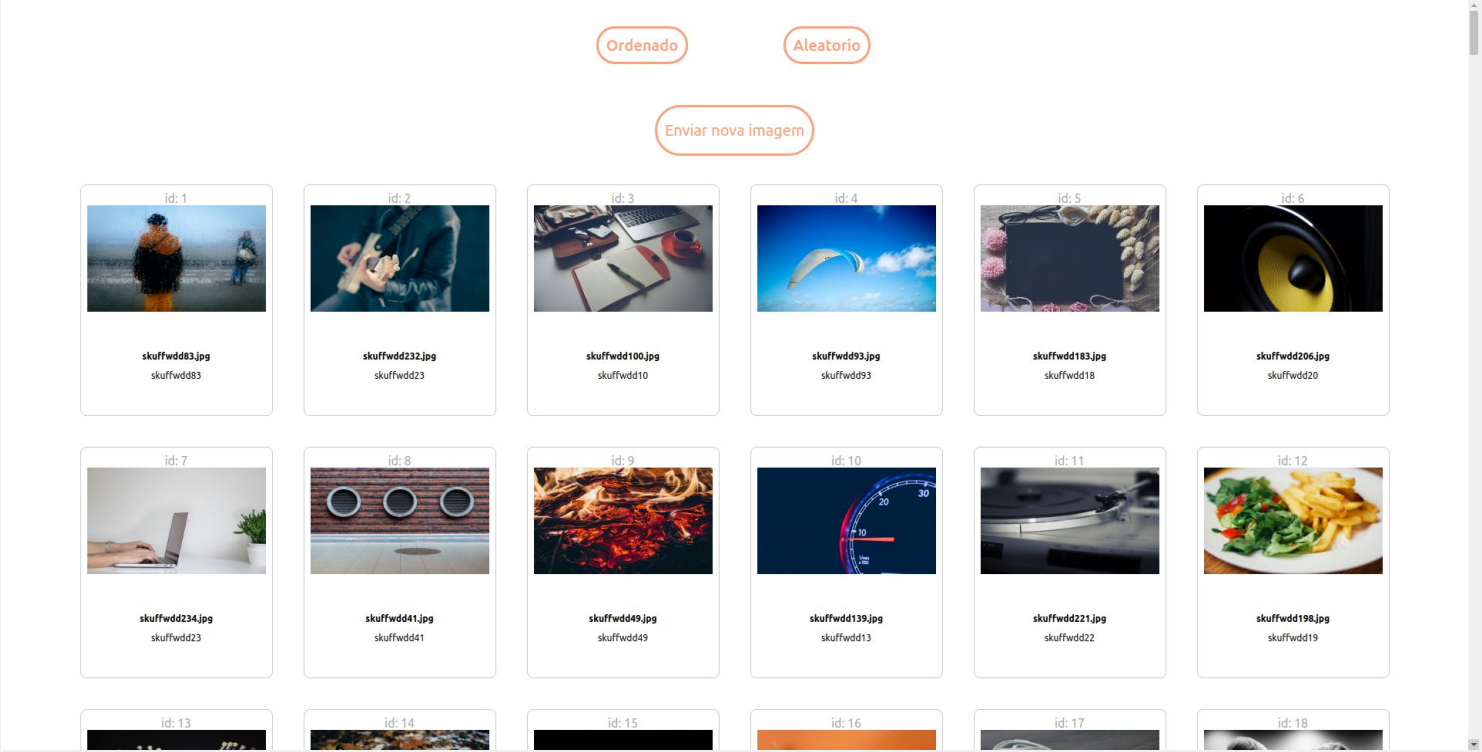
"Quando o conceito de excelência for intrínseco ao dev, naturalmente ele precisará estudar algoritmos e estruturas de dados"

Clarice Lispector

Vamos ver alguns exemplos…

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

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



Exemplo

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

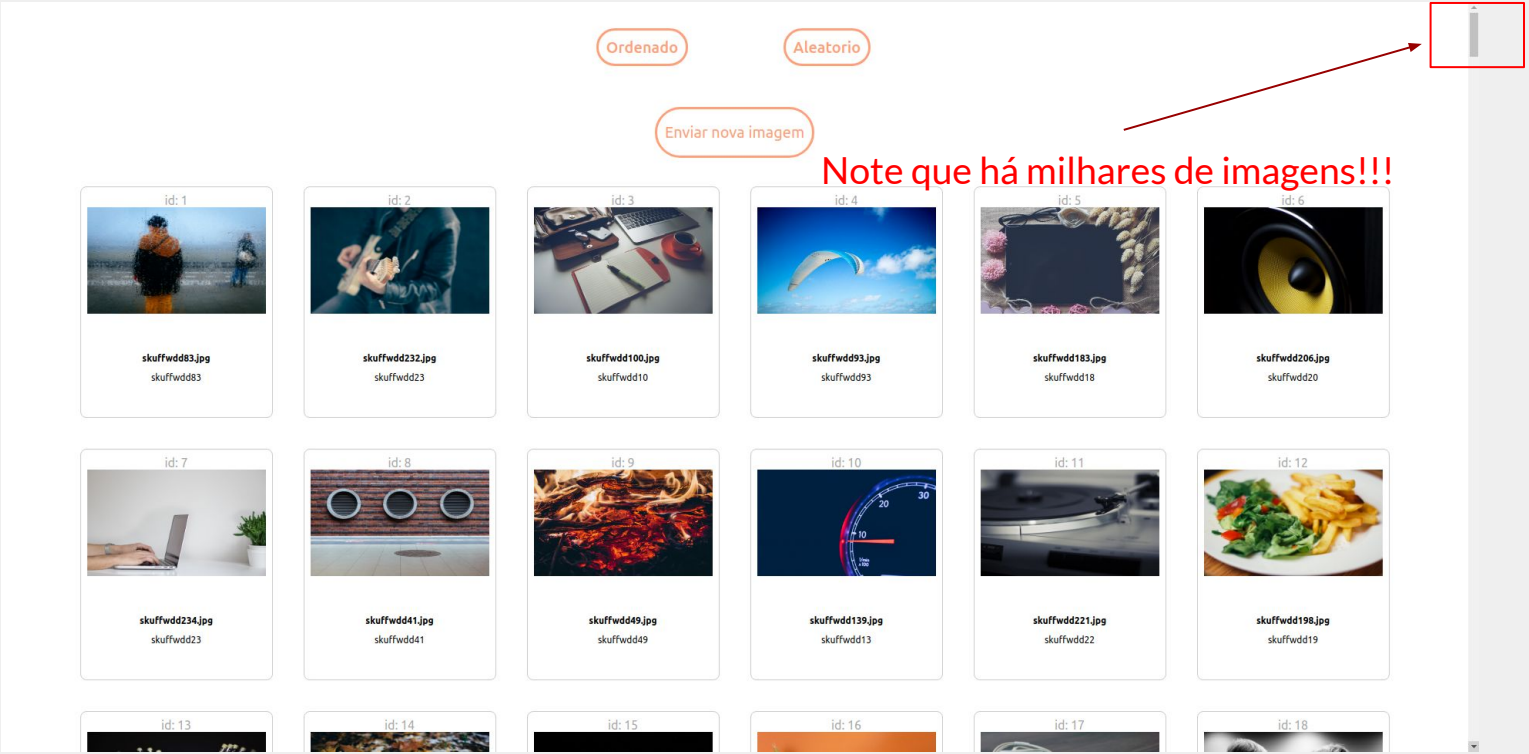
O usuário deseja pesquisar o id 1 no array; return → 0

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

O usuário deseja pesquisar o id 7 no array; return → 3

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

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



Busca sequencial

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

Busca sequencial - Complexidade: O(N) (lê-se “Ó de n”)

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

Ó de quê? O que é isso?

Ok… Precisamos dar um passo para trás...

Complexidade de algoritmos

Complexidade de algoritmos

Definição informal (logo, menos precisa): 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.

Complexidade de algoritmos

Uma maneira muito direta de calcular a complexidade seria encontrando alguma fórmula que dê o número exato de operações feitas pelo algoritmo para chegar no resultado, em função do tamanho da entrada. Por exemplo, no algoritmo

```
for(i=0; i<N; i++) print(i);
```

poderíamos dizer que o tempo gasto é:

T(N) =
    N*(tempo gasto por uma comparação entre i e N) +
    N*(tempo gasto para incrementar i) +
    N*(tempo gasto por um print)

Complexidade de algoritmos

No entanto, dá muito trabalho fazer uma conta super precisa dessas e geralmente nem vale a pena. Por exemplo, suponha que tenhamos nos esforçado bastante e descoberto que um certo algoritmo gasta tempo

$T(N) = 10*N^2 + 137*N + 15$

Complexidade de algoritmos

No entanto, dá muito trabalho fazer uma conta super precisa dessas e geralmente nem vale a pena. Por exemplo, suponha que tenhamos nos esforçado bastante e descoberto que um certo algoritmo gasta tempo

$T(N) = 10*N^2 + 137*N + 15$

Nesse caso o termo quadrático $10*N^2$ é mais importante que os outros pois para praticamente qualquer valor de N ele irá dominar o total da soma.

Complexidade de algoritmos

No entanto, dá muito trabalho fazer uma conta super precisa dessas e geralmente nem vale a pena. Por exemplo, suponha que tenhamos nos esforçado bastante e descoberto que um certo algoritmo gasta tempo

$T(N) = 10*N^2 + 137*N + 15$

Nesse caso o termo quadrático $10*N^2$ é mais importante que os outros pois para praticamente qualquer valor de N ele irá dominar o total da soma.

Para fins de estimativa poderíamos simplificar a fórmula para $T(N) = 10*N^2$ sem perder muita coisa.

Complexidade de algoritmos

No entanto, dá muito trabalho fazer uma conta super precisa dessas e geralmente nem vale a pena. Por exemplo, suponha que tenhamos nos esforçado bastante e descoberto que um certo algoritmo gasta tempo

$T(N) = 10*N^2 + 137*N + 15$

Nesse caso o termo quadrático $10*N^2$ é mais importante que os outros pois para praticamente qualquer valor de N ele irá dominar o total da soma.

Para fins de estimativa poderíamos simplificar a fórmula para $T(N) = 10*N^2$ sem perder muita coisa.

Outro ponto em que podemos simplificar a nossa fórmula é o fator constante multiplicando o N^2. Para prever o quão rápido o tempo de execução cresce dependendo da entrada não importa se $T(N) = 10*N$ ou $T(N) = 1000*N$; em ambos os casos dobrar o N vai quadruplicar o tempo de execução.

Notação Big O

Notação Big O

“Dadas duas funções f e g, dizemos que f ∈ O(g) se existem constantes x0 e c tal que para todo x > x0 vale f(x) < c*g(x)”

Notação Big O

“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)”

É importante sinalizar que essa é uma definição mais teórica, raramente é usada de fato no "mundo real"...

Notação Big O

“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 $f(x) < c*g(x)$”

É importante sinalizar que essa é uma definição mais teórica, raramente é usada de fato no "mundo real"... Bom, de nada adianta uma informação precisa quando ninguém consegue entendê-la.

Notação Big O

Definição informal (logo, menos precisa): Queremos saber o comportamento do algoritmo para N suficientemente grande, isso implica em saber quais termos que crescem mais rápido, ou seja, os que dominem o valor total.

Notação Big O

Definição informal (logo, menos precisa): Queremos saber o comportamento do algoritmo para N suficientemente grande, isso implica em saber quais termos que crescem mais rápido, ou seja, os que dominem o valor total.

Exemplo: $T(N) = 10*N^2 + 137*N + 15$ (de um slide anterior)

Notação Big O

Definição informal (logo, menos precisa): Queremos saber o comportamento do algoritmo para N suficientemente grande, isso implica em saber quais termos que crescem mais rápido, ou seja, os que dominem o valor total.

Exemplo: $T(N) = 10*N^2 + 137*N + 15$ (de um slide anterior)

O termo que cresce mais rápido é o “$10*N^2$”.

Notação Big O

Definição informal (logo, menos precisa): Queremos saber o comportamento do algoritmo para N suficientemente grande, isso implica em saber quais termos que crescem mais rápido, ou seja, os que dominem o valor total.

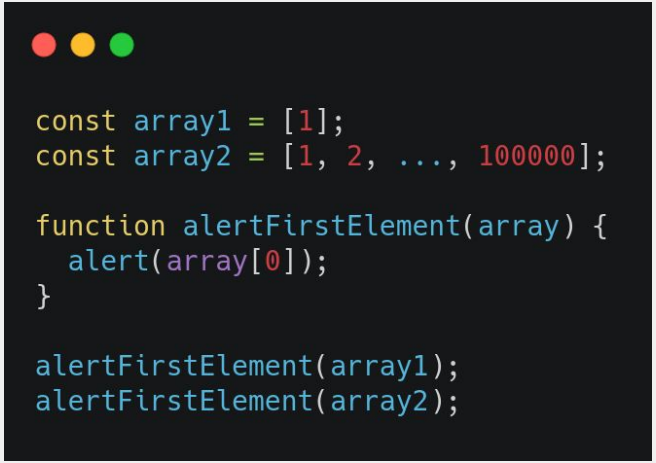
Exemplo: $T(N) = 10*N^2 + 137*N + 15$ (de um slide anterior)

O termo que cresce mais rápido é o “$10*N^2$”.

Também despreza-se a constante! Logo: $O(N^2)$

Notação Big O - Exemplo

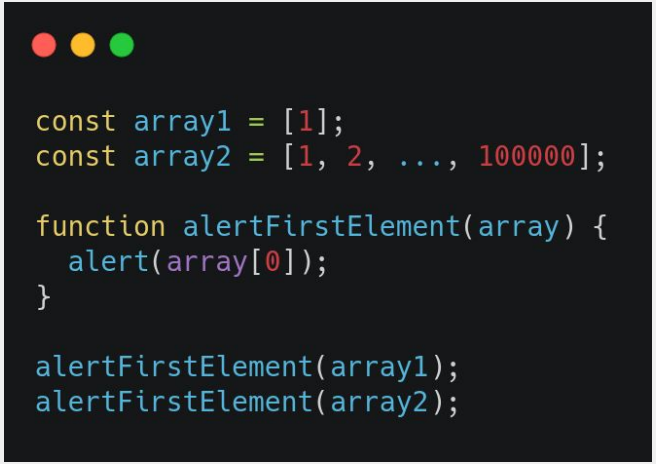
Considere o seguinte:



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

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

alertFirstElement(array1);
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?



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

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

alertFirstElement(array1);
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?

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

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

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

E se substituirmos a função acima por esta?

```
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);
```

```
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.

Notação Big O

Um grande número confunde a gente, ser humano. Raciocinar sobre números grandes é ainda mais difícil. Então, como podemos tornar a análise de algoritmos mais perspicaz? Medir coisas e aplicar escalas ajuda. **A visualização definitivamente ajuda.**

Notação Big O

Um grande número confunde a gente, ser humano. Raciocinar sobre números grandes é ainda mais difícil. Então, como podemos tornar a análise de algoritmos mais perspicaz? Medir coisas e aplicar escalas ajuda. **A visualização definitivamente ajuda.**

Digamos que temos um determinado algoritmo. Nós o executamos em um número modesto de cinquenta elementos. Nosso algoritmo possui implementações com diferentes complexidades de tempo. Quão eficiente é cada algoritmo? Quão bem ele se expande quando aumentamos o número de elementos entre um e cinquenta? Essas são as perguntas com as quais as empresas de tecnologia de sucesso se preocupam ao expandir produtos ou serviços para milhares, milhões ou até bilhões de usuários.

Notação Big O

Um grande número confunde a gente, ser humano. Raciocinar sobre números grandes é ainda mais difícil. Então, como podemos tornar a análise de algoritmos mais perspicaz? Medir coisas e aplicar escalas ajuda. **A visualização definitivamente ajuda.**

Digamos que temos um determinado algoritmo. Nós o executamos em um número modesto de cinquenta elementos. Nosso algoritmo possui implementações com diferentes complexidades de tempo. Quão eficiente é cada algoritmo? Quão bem ele se expande quando aumentamos o número de elementos entre um e cinquenta? Essas são as perguntas com as quais as empresas de tecnologia de sucesso se preocupam ao expandir produtos ou serviços para milhares, milhões ou até bilhões de usuários.

Não pode melhorar o que não se pode medir!!

Em resumo

Notação Big O

A notação não calcula o tempo de execução exata de um algoritmo (por exemplo, “esta função vai terminar a execução em 400 ms”), mas sim a taxa na qual seu tempo de execução cresce quando o tamanho de sua entrada aumenta.

Notação Big O

A notação não calcula o tempo de execução exata de um algoritmo (por exemplo, “esta função vai terminar a execução em 400 ms”), mas sim a taxa na qual seu tempo de execução cresce quando o tamanho de sua entrada aumenta.

É muito comum chamar de Big-O quando a informação é sobre a média de execução, mesmo que isto não seja a definição mais precisa para ele. É comum também fazer aproximações ou desconsiderar casos excepcionais.

Notação Big O

A notação não calcula o tempo de execução exata de um algoritmo (por exemplo, “esta função vai terminar a execução em 400 ms”), mas sim a taxa na qual seu tempo de execução cresce quando o tamanho de sua entrada aumenta.

É muito comum chamar de Big-O quando a informação é sobre a média de execução, mesmo que isto não seja a definição mais precisa para ele. É comum também fazer aproximações ou desconsiderar casos excepcionais.

Por exemplo, um hash costuma ter o pior caso como O(N), mas em geral é dito que ele tem O(1) porque isto é o que geralmente acontece. O caso O(N) acontece apenas em casos extremos.

Notação Big O

A notação não calcula o tempo de execução exata de um algoritmo (por exemplo, “esta função vai terminar a execução em 400 ms”), mas sim a taxa na qual seu tempo de execução cresce quando o tamanho de sua entrada aumenta.

É muito comum chamar de Big-O quando a informação é sobre a média de execução, mesmo que isto não seja a definição mais precisa para ele. É comum também fazer aproximações ou desconsiderar casos excepcionais.

Por exemplo, um hash costuma ter o pior caso como O(N), mas em geral é dito que ele tem O(1) porque isto é o que geralmente acontece. O caso O(N) acontece apenas em casos extremos.

Em muitos casos é melhor ater-se ao caso médio ainda que seria importante saber antes se realmente há boa distribuição para garantir que a média ocorra na prática. Então o mais comum é usarmos o caso típico como referência e não o pior caso.

Os Casos mais comuns

Notação Big O

- O(1) - Tempo constante
- O(logN) - Tempo logarítmico
- O(N) - Tempo linear
- O(NlogN) - Tempo linear~quadrático
- $O(N^2)$ - Tempo quadrático

Notação Big O

- O(1)
- O(logN)
- O(N)
- O(NlogN)
- O(N^2)

Lembrando que quanto mais acima estiver nessa lista, melhor é a complexidade!

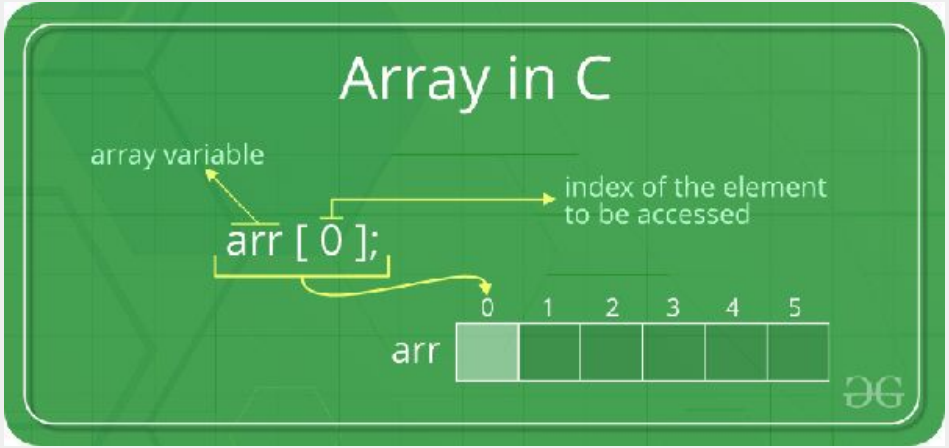
Notação Big O

- O(1)
- O(logN)
- O(N)
- O(NlogN)
- $O(N^2)$

Notação Big O

- O(1)

  Exemplos: Um array e uma tabela hash (ou dicionário) são exemplos de estruturas de dados que possuem acesso constante.

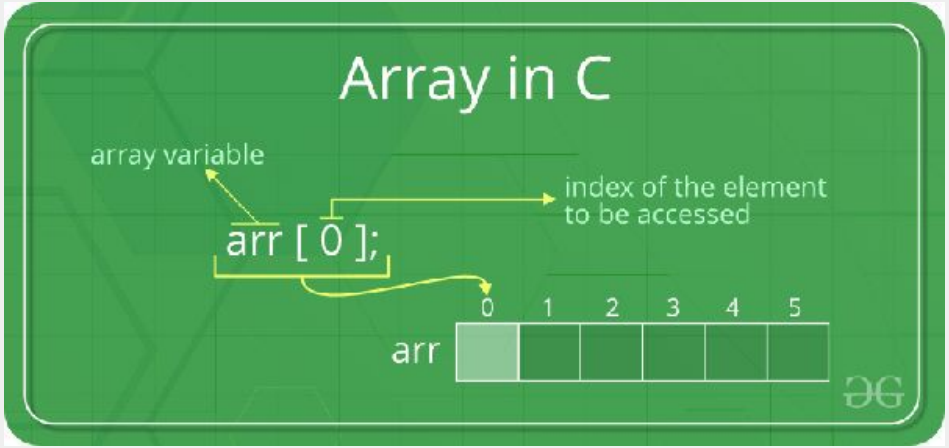


Notação Big O

- O(1)

  Exemplos: Um array e uma tabela hash (ou dicionário) são exemplos de estruturas de dados que possuem acesso constante.

Acesso direto!



Notação Big O

- O(1)
- O(logN)
- O(N)
- O(NlogN)
- $O(N^2)$

Notação Big O

- O(1)
- O(logN)

  Exemplo: busca binária

Notação Big O

- O(1)
- O(logN)

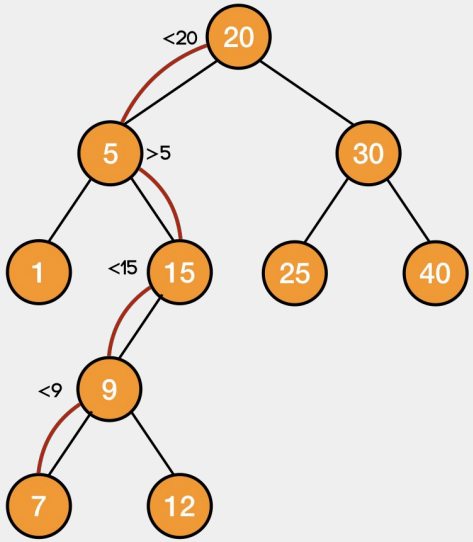
  Exemplo: busca binária

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(logN)

Outro exemplo: search em uma árvore binária de busca balanceada



Searching in BST

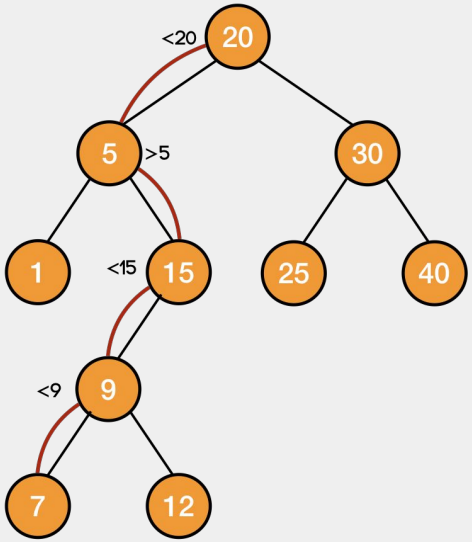
SEARCH FOR 7
STEP 1: 7 IS SMALLER THAN 20: GO LEFT
STEP 2: 7 IS GREATER THAN 5: GO RIGHT
STEP 3: 7 IS SMALLER THAN 15: GO LEFT
STEP 4: 7 IS SMALLER THAN 9: GO LEFT

Notação Big O

- O(1)
- O(logN)

Outro exemplo: search em uma árvore binária de busca balanceada

No pior caso, a quantidade de passos será o tamanho da árvore. E a altura da árvore é: altura = log(N), onde N é a quantidade de nós.



Searching in BST

SEARCH FOR 7
STEP 1: 7 IS SMALLER THAN 20: GO LEFT
STEP 2: 7 IS GREATER THAN 5: GO RIGHT
STEP 3: 7 IS SMALLER THAN 15: GO LEFT
STEP 4: 7 IS SMALLER THAN 9: GO LEFT

Notação Big O

- O(1)
- O(logN)
- O(N)
- O(NlogN)
- $O(N^2)$

Notação Big O

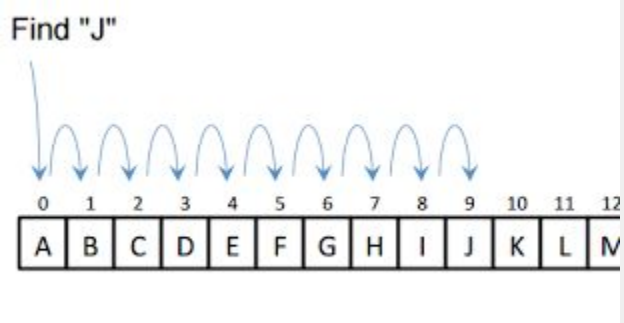
- O(1)
- O(logN)
- O(N)

Já vimos lá atrás… Alguém saberia dar um exemplo?

Notação Big O

- O(1)
- O(logN)
- O(N)

Exemplo: busca sequencial.

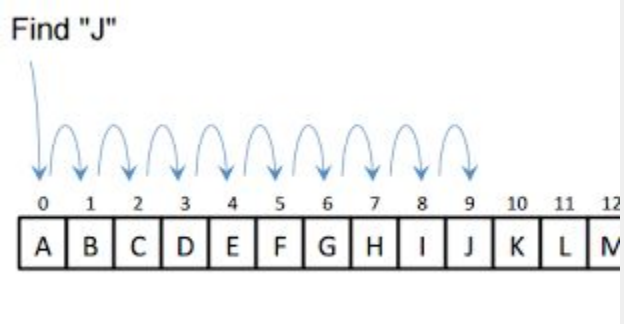


Notação Big O

- O(1)
- O(logN)
- O(N)

  Exemplo: busca sequencial.

Busca um a um! Logo, no pior caso precisamos percorrer os n elementos do array. logo: O(N).



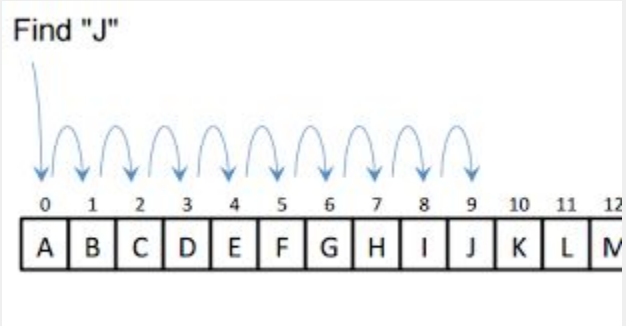

```
function loopN(arr) {
  for (let i = 0; i < arr.length; i++)
    console.log(i);
}
```

Notação Big O

- O(1)
- O(logN)
- O(N)

  Exemplo: busca sequencial.

Busca um a um! Logo, no pior caso precisamos percorrer os n elementos do array. logo: O(N).



Um único loop!

```
function loopN(arr) {
  for (let i = 0; i < arr.length; i++)
    console.log(i);
}
```

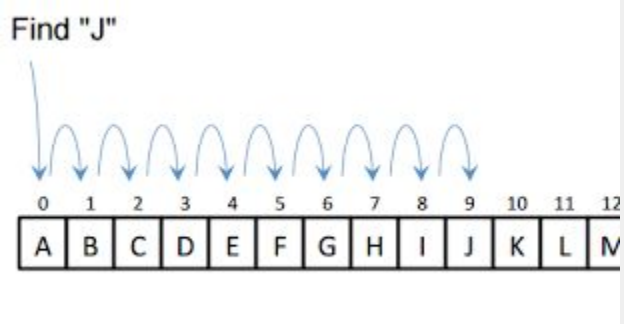
Notação Big O

- O(1)
- O(logN)
- O(N)

  Exemplo: busca sequencial.

E se forem 2 loops, qual seria a complexidade?

Busca um a um! Logo, no pior caso precisamos percorrer os n elementos do array. logo: O(N).




```
function loopN(arr) {
  for (let i = 0; i < arr.length; i++)
    for (let j = i; j < arr.length; j++)
      console.log(i + j);
}
```

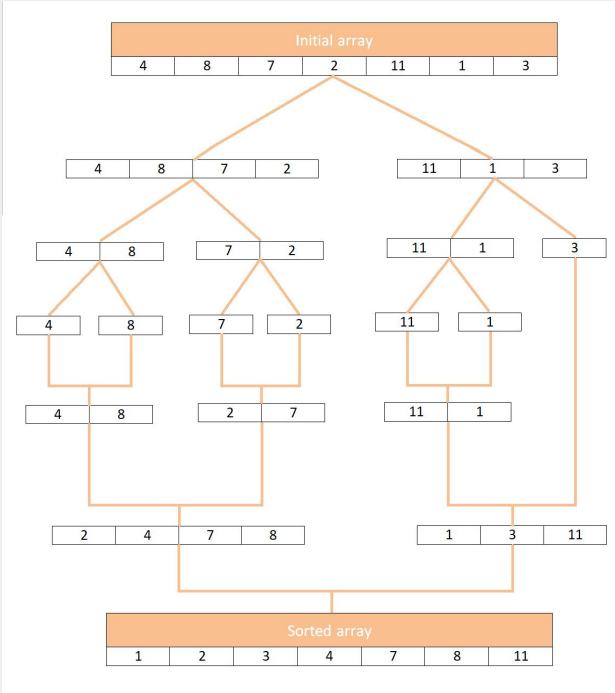
Notação Big O

- O(1)
- O(logN)
- O(N)
- O(NlogN)
- $O(N^2)$

Notação Big O

- O(1)
- O(logN)
- O(N)
- O(NlogN)

Exemplo: merge sort.



Notação Big O

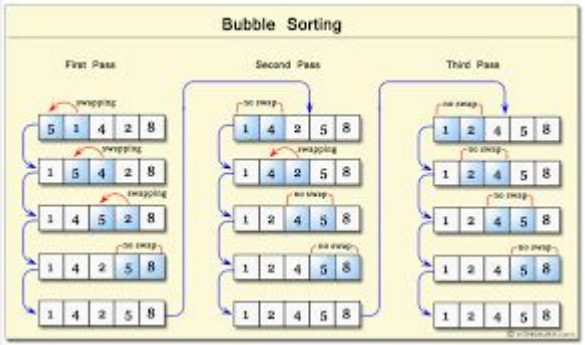
- O(1)
- O(logN)
- O(N)
- O(NlogN)
- $O(N^2)$

Notação Big O

- O(1)
- O(logN)
- O(N)
- O(NlogN)
- O(N²)

Exemplo: bubble sort

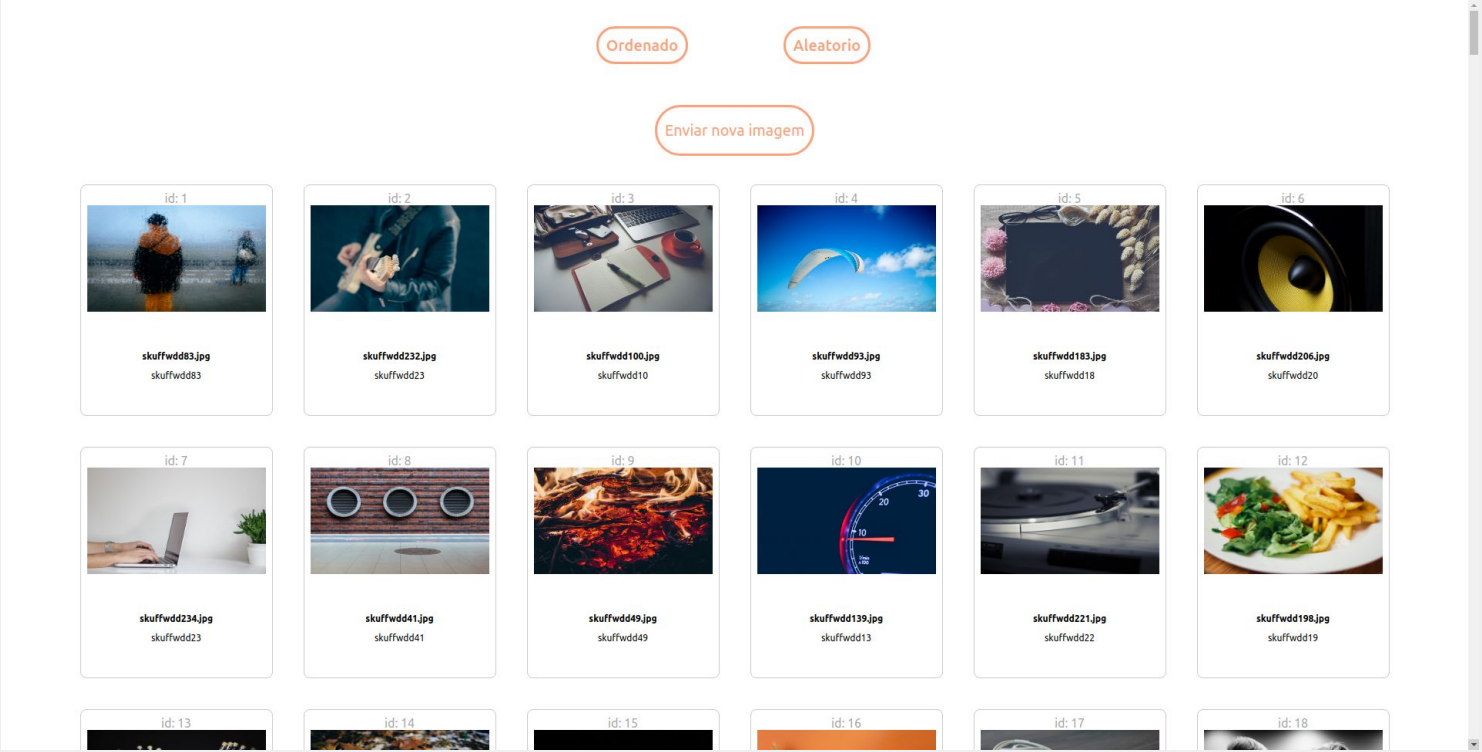
6 5 3 1 8 7 2 4



Recapitulando o exemplo...

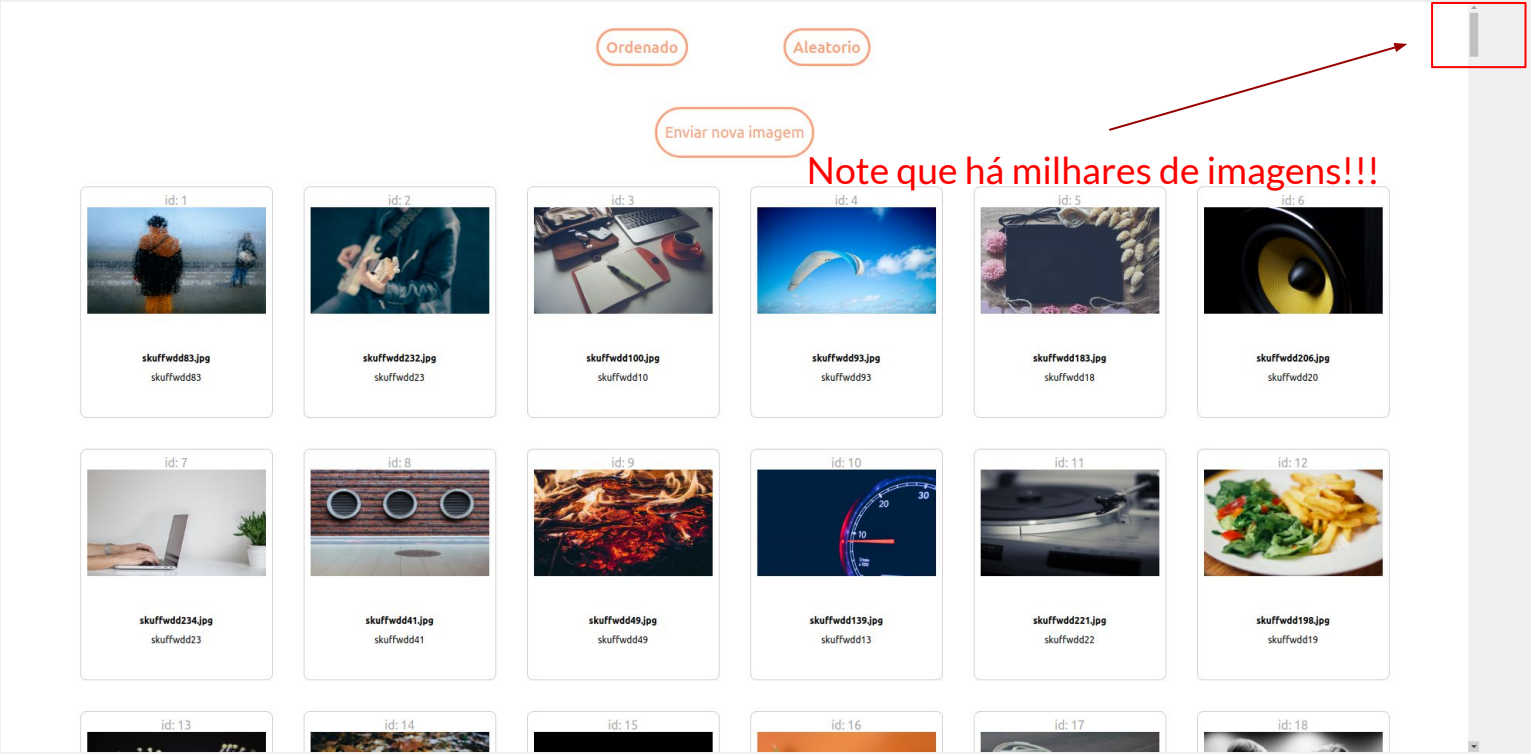
Use case: o usuário deseja poder pesquisar o card pelo id.

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



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

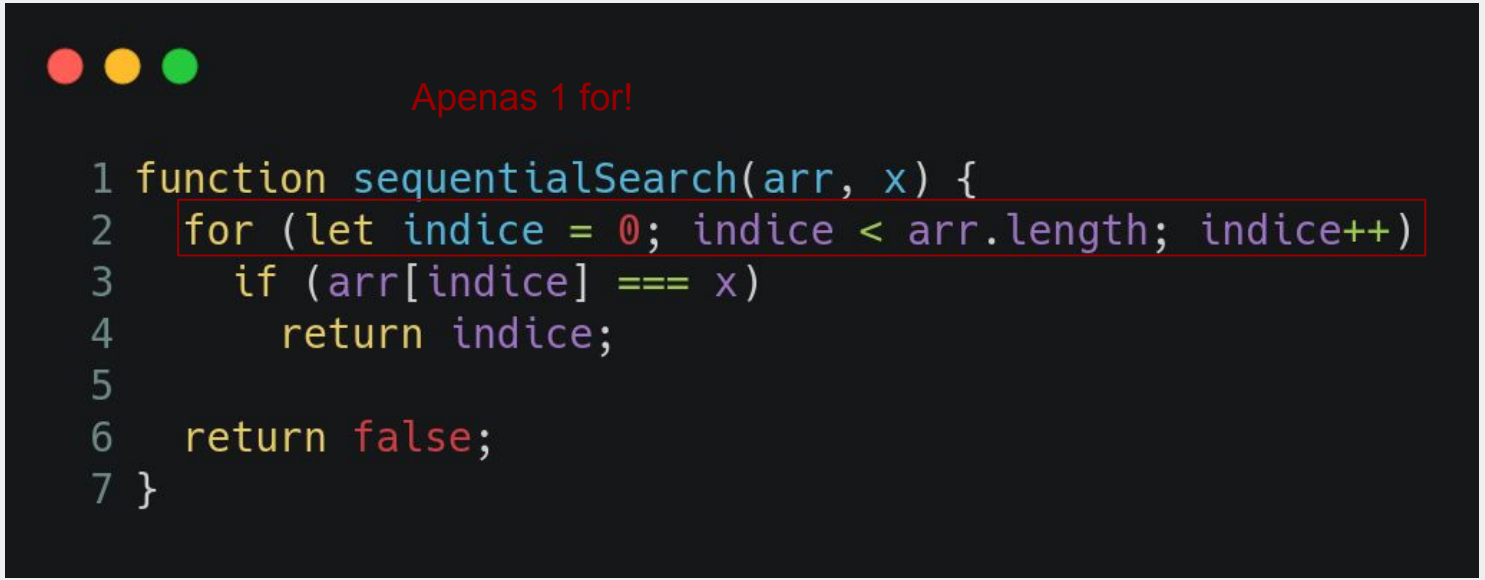
*É garantido que os cards estão ordenados pelo id. O que o front precisa fazer é retornar o índice do id no array (se existir).



Busca sequencial - Complexidade: O(N)

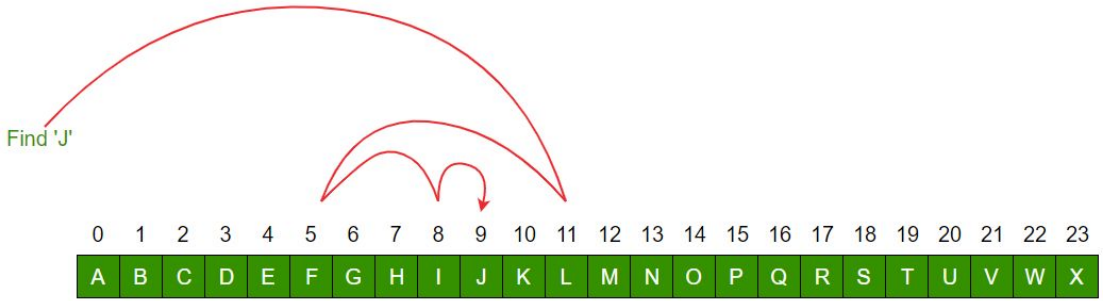
```
1 function sequentialSearch(arr, x) {
2   for (let indice = 0; indice < arr.length; indice++)
3     if (arr[indice] === x)
4       return indice;
5
6   return false;
7 }
```

Busca sequencial - Complexidade: O(N)



Com base no que a gente já viu até agora, alguma ideia de como resolver esse problema de uma forma otimizada?

Busca binária!



Busca binária - O(logN)

```
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);
}
```

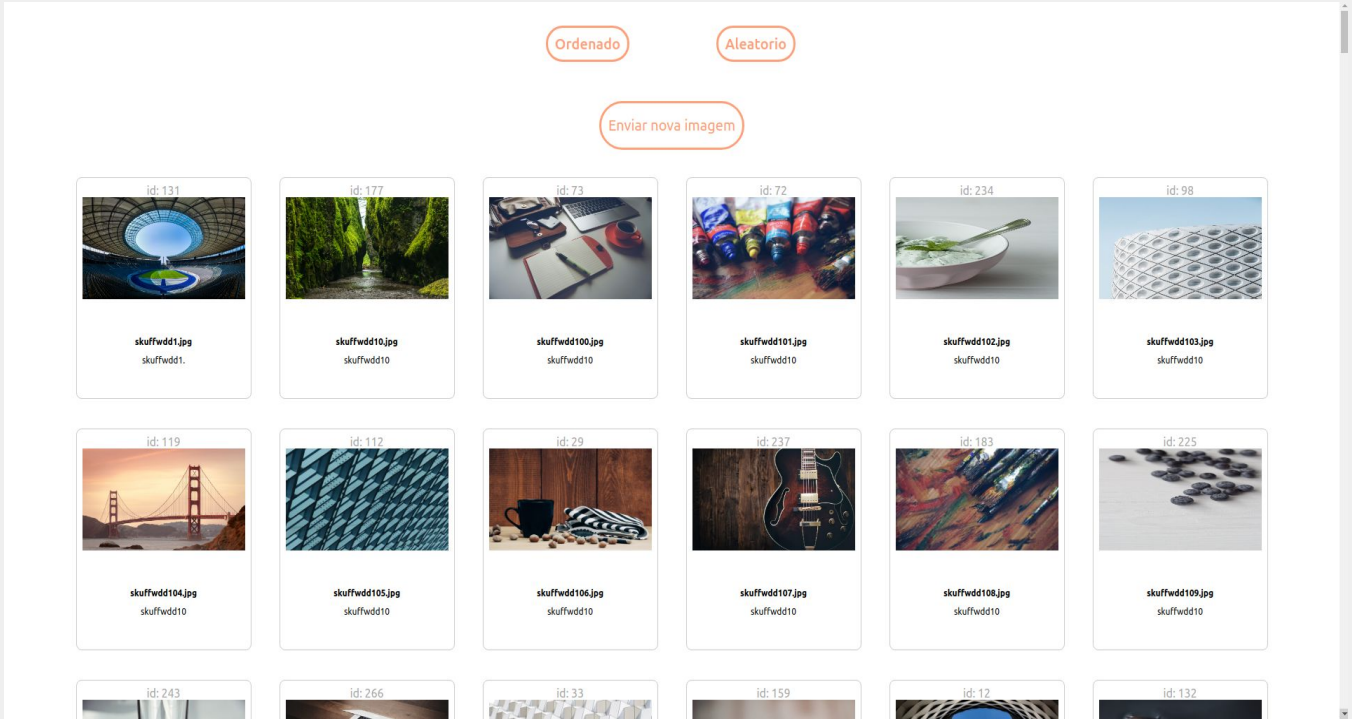
Pensou no método find? (Array.prototype.find(x))

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

Outro exemplo...

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

Seja os ids

[3, 0, 5, 1, 4]

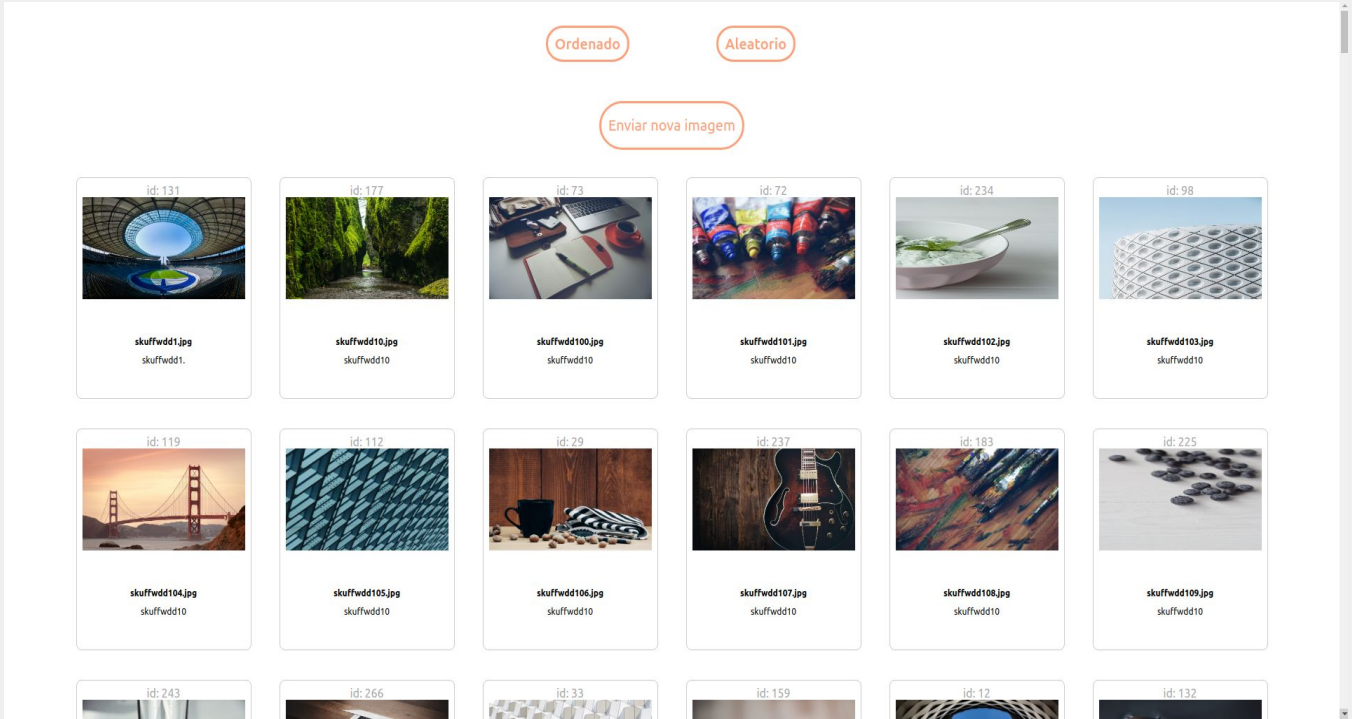
O menor id que não está listado é o 2.

*Note que o menor id possível é 0.

**No mundo real, geralmente o responsável por gerar o menor id seria o back, pois se não poderá cair num problema de id esparso, degradando a performance do back-end. Mas por algum problema, de natureza diversa (como falta de conhecimento no back-end), o front além de enviar o blob da imagem para o back-end, também precisará enviar o menor id.

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 - Força bruta

No pior caso todos os ids estão preenchidos, nesse caso o menor id é o tamanho do array mais um.

Exemplo:

[3,1,4,2] → menor é 5.

Gerando ideias - Força bruta

No pior caso todos os ids estão preenchidos, nesse caso o menor id é o tamanho do array mais um.

Exemplo:

[3,1,4,2] → menor é 5.

Então, podemos primeiro procurar o id 1, se não existir retorna 1. Se existir, procuramos o id 2 e assim por diante...

Solução - Força bruta

```
function findLowerId(arr) {
  let { length } = arr;

  for (let id = 0; id < length; id++) {
    let found = false;

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

    if (!found) return id;
  }

  return length + 1;
}
```

Solução - Força bruta

Qual a complexidade?

```
function findLowerId(arr) {
  let { length } = arr;

  for (let id = 0; id < length; id++) {
    let found = false;

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

    if (!found) return id;
  }

  return length + 1;
}
```

Solução - Força bruta

Qual a complexidade?

$O(n^2)$!

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

```
function findLowerId(arr) {
  let { length } = arr;

  for (let id = 0; id < length; id++) {
    let found = false;

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

    if (!found) return id;
  }

  return length + 1;
}
```

Alguma forma otimizada?

Gerando ideias - Sort

Se ordenarmos o vetor, podemos fazer uma busca sequencial!

Gerando ideias - Sort

```
function findLowerId(arr) {
  let { length } = arr;
  let sortedVector = arr.sort((a, b) => a - b);

  for (let id = 0; id < length; id++)
    if (sortedVector[id] !== id) return id;

  return length + 1;
}
```

Gerando ideias - Sort

Qual a complexidade?

```
function findLowerId(arr) {
  let { length } = arr;
  let sortedVector = arr.sort((a, b) => a - b);

  for (let id = 0; id < length; id++)
    if (sortedVector[id] !== id) return id;

  return length + 1;
}
```

Gerando ideias - Sort

Qual a complexidade?

```
function findLowerId(arr) {
  let { length } = arr;
  let sortedVector = arr.sort((a, b) => a - b);

  for (let id = 0; id < length; id++)
    if (sortedVector[id] !== id) return id;

  return length + 1;
}
```

O(N)? 😄

Gerando ideias - Sort

Qual a complexidade?

```
function findLowerId(arr) {
  let { length } = arr;
  let sortedVector = arr.sort((a, b) => a - b);

  for (let id = 0; id < length; id++)
    if (sortedVector[id] !== id) return id;

  return length + 1;
}
```

Ops... Não podemos esquecer do sort!

Gerando ideias - Sort

Qual a complexidade?

O(NlogN + N)

```
function findLowerId(arr) {
  let { length } = arr;
  let sortedVector = arr.sort((a, b) => a - b);

  for (let id = 0; id < length; id++)
    if (sortedVector[id] !== id) return id;

  return length + 1;
}
```

Gerando ideias - Sort

Qual a complexidade?

O(NlogN)

```
function findLowerId(arr) {
  let { length } = arr;
  let sortedVector = arr.sort((a, b) => a - b);

  for (let id = 0; id < length; id++)
    if (sortedVector[id] !== id) return id;

  return length + 1;
}
```

Predominante!!

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)?

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) {
  const { length } = arr;

  for (let id = 0; id < length; id++)
    if(!struct.has(id)) return id;

  return length + 1;
}
```

Gerando ideias

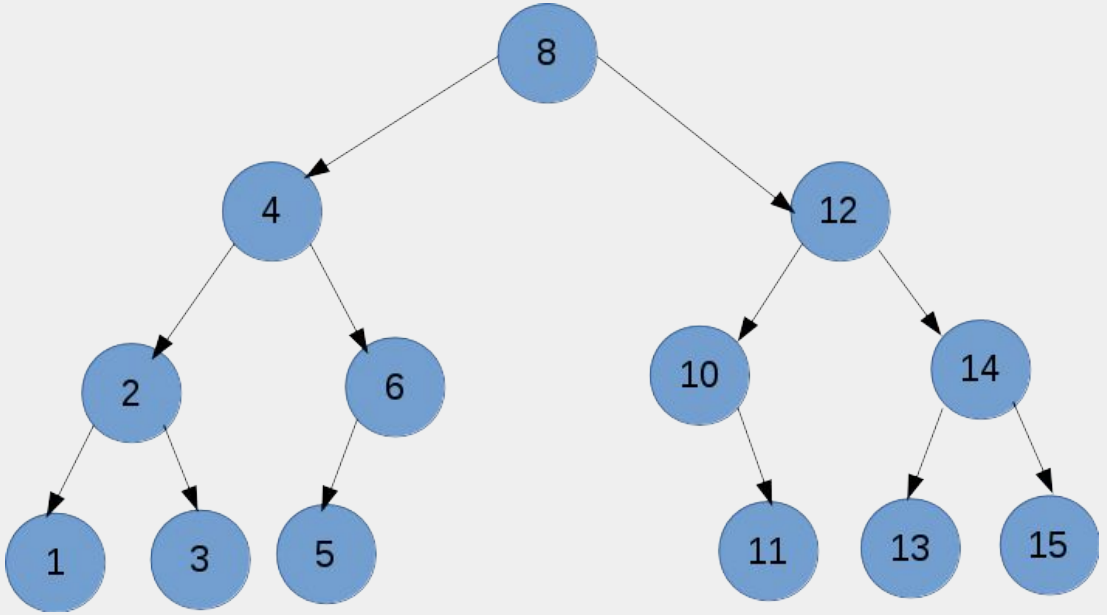
Vimos que o gargalo é no sort… Há uma forma de saber se o id existe em O(1)?

Algo como:

Seria perfeito pois a complexidade seria O(N)!!

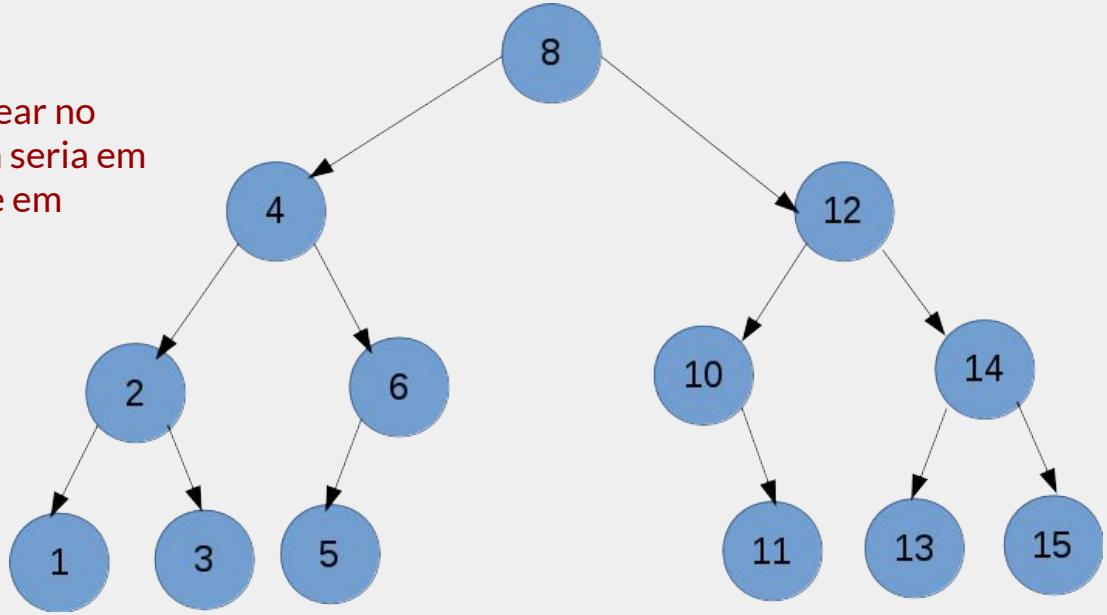
```
function findLowerId (arr) {
  const { length } = arr;

  for (let id = 0; id < length; id++)
    if(!struct.has(id)) return id;

  return length + 1;
}
```

Gerando ideias - Árvore binária



Gerando ideias - Árvore binária

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

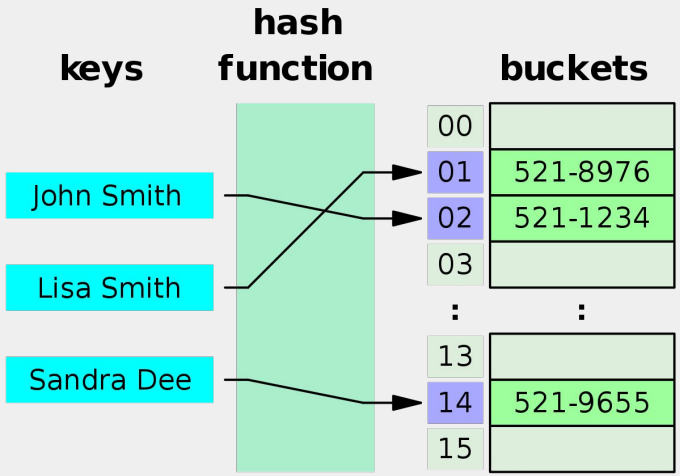


Queremos saber se um determinado id existe em O(1)!!!

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

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

Solução - Hash table

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

Solução - Hash table

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

Solução - Hash table

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

Solução - Hash table

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

Solução - Hash table

Qual a complexidade?

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

Solução - Hash table

Qual a complexidade?

O(N + N) = O(2N)

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

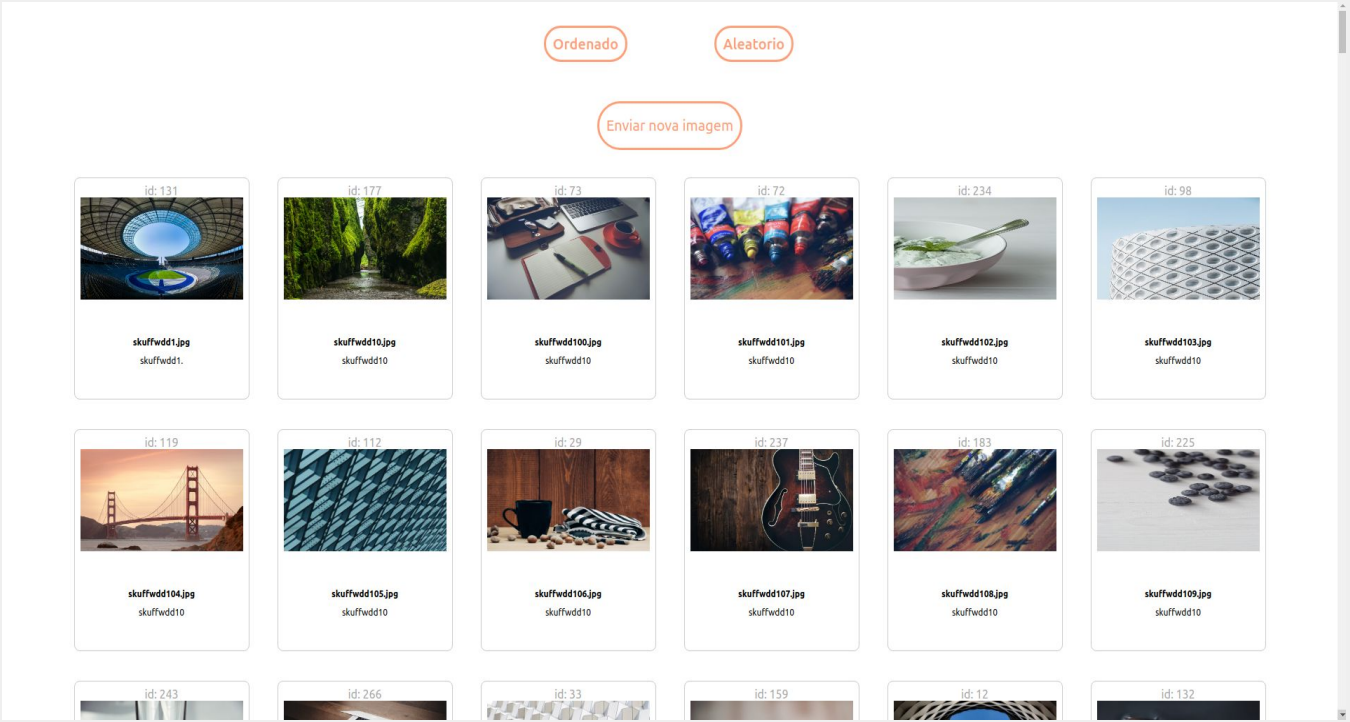
Solução - Hash table

Qual a complexidade?

O(N)

```
function findLowerId (arr) {
  const hashTable = new Map();
  const { length } = arr;

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

  for (let id = 0; id < length; id++)
    if(!hashTable.has(id)) return id;

  return length + 1;
}
```

Bônus - Use case: como encontrar o maior id?



Primeiro faz um Sort e depois pega o último elemento?

Primeiro faz um Sort e depois pega o último elemento?

Como vimos, a complexidade fica: O(NlogN)

Cuidados nas soluções “rápidas” !!
Sempre é bom pensar mais um pouco...

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

Overview

- Esse assunto não é de interesse só do back ou de uma perspectiva de back. O que acontece, é que esse nível de "preocupação" é mais visto no back-end, mas o front também precisa se preocupar!

Overview

- Esse assunto não é de interesse só do back ou de uma perspectiva de back. O que acontece, é que esse nível de "preocupação" é mais visto no back-end, mas o front também precisa se preocupar!
- Lembre-se que fazer o usuário esperar pode implicar em perder dinheiro! Principalmente em um cenário como o da black friday.

Overview

- Esse assunto não é de interesse só do back ou de uma perspectiva de back. O que acontece, é que esse nível de "preocupação" é mais visto no back-end, mas o front também precisa se preocupar!
- Lembre-se que fazer o usuário esperar pode implicar em perder dinheiro! Principalmente em um cenário como o da black friday.
- Além de:

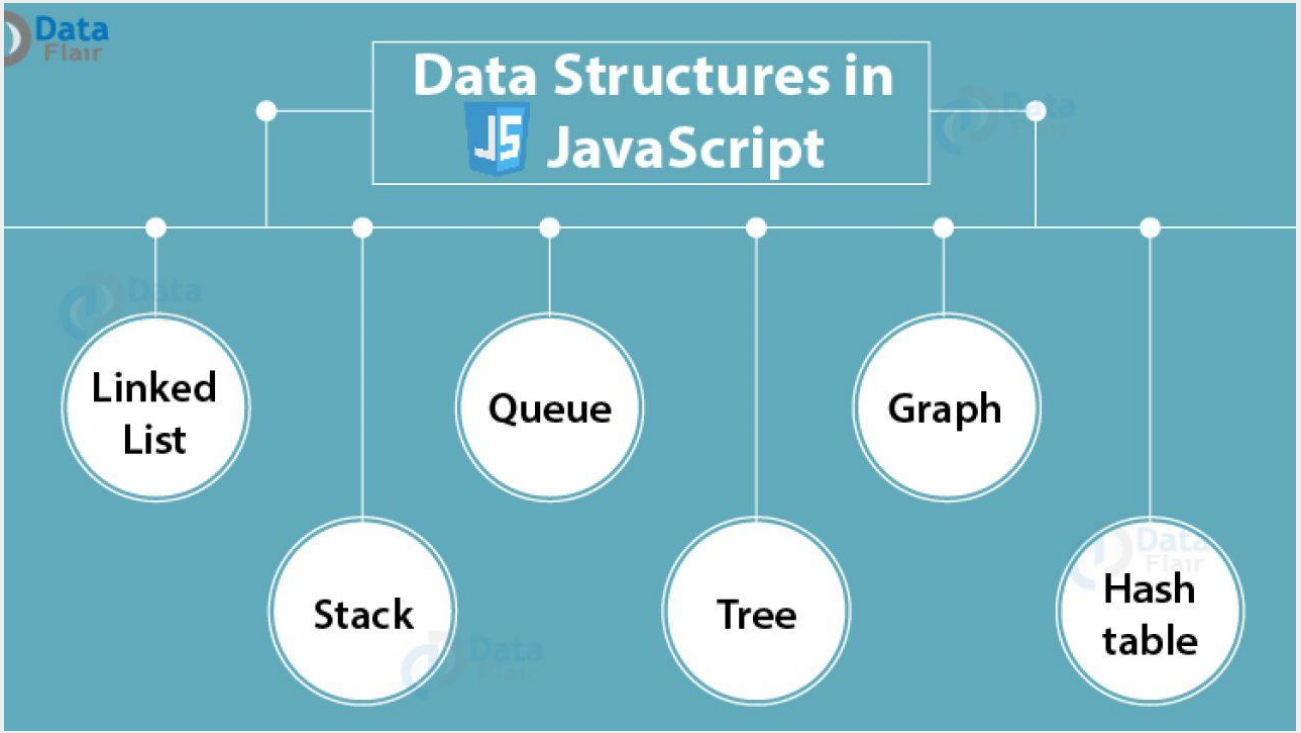
"Quando o conceito de excelência for intrínseco ao dev, naturalmente ele precisará se **preocupar** com algoritmos e estruturas de dados"

Clarice Lispector + Gustavo Oliveira

gustavooliveiraf

data-structures-and-algorithms

JavaScript



Fim!

Obrigado!

Gustavo Oliveira
Engenheiro de Software na Gympass



gustavooliveiraf

Dúvidas?

Bônus - Para pensar um pouco mais...
Use case: como encontrar os k maiores ids?

Bônus - Para pensar um pouco mais...
Use case: como encontrar os k maiores ids?

Dica: a complexidade fica em O(nlogk) :)