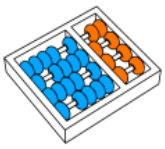

Análise de desempenho de funções nativas do JavaScript

A. Diamint *G. Higa* *L. F. Bittencourt*

Relatório Técnico - IC-PFG-22-09
Projeto Final de Graduação
2022 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Análise de desempenho de funções nativas do Javascript

Agustina Diamint, Guilherme Higa, Luiz Fernando Bittencourt

Resumo. Este trabalho busca avaliar o desempenho de execução da implementação da biblioteca Node.js de algumas funções nativas do Javascript, pertencentes aos objetos *String* e *Array*. As funções do objeto String testadas foram: *match(), substring() e toUpperCase()*. As funções do objeto *Array* testadas foram: *concat()*, *find()*, *reverse()*, *shift()*, *unshift()*, *sort()* e *push()*.

Para realizar a avaliação, utilizamos a biblioteca ***benchmark.js***, que realiza a execução dos blocos de códigos a serem testados em cima dos *datasets* fornecidos por uma quantidade de vezes determinada via parâmetro (no caso deste trabalho, em torno de 90 execuções por dataset). Ao final do teste, a biblioteca fornece um conjunto de dados estatísticos relevantes relacionados ao tempo de execução do programa, como tempo médio de execução, margem de erro do tempo médio e porcentagem relativa de erro. Para este trabalho, utilizamos apenas o tempo médio de execução.

Utilizamos datasets de tamanhos variados, com valores tanto aleatórios quanto pensados especialmente para testar casos de borda dos algoritmos. Além dos algoritmos nativos, testamos o desempenho de implementações manuais de algoritmos semelhantes com complexidades conhecidas a fim de comparar o desempenho do algoritmo nativo com o implementado.

De posse dos dados, foram produzidos gráficos de desempenho para os algoritmos usando como pontos o tempo de execução médio necessário para a completude de execução do programa em cima de cada *dataset*.

Palavras-Chave: Análise de Complexidade; Javascript, Array, String.

Índice

1. Introdução

Informações sobre a complexidade de implementação das funções nativas do Javascript são limitadas na literatura e documentação. Por outro lado, o conhecimento deste dado é de extrema importância para os usuários dessa linguagem. Levando em consideração o fato de que esta é a linguagem mais utilizada para a construção de páginas e aplicações *web* do mundo[1], e de que métricas de desempenho como First Contentful Paint (FCP), First Input Delay (FID) e Time to Interactive (TTI) impactam diretamente o interesse do usuário como taxa de rejeição e páginas lidas por sessão[2][3], consequentemente, no resultado financeiro das empresas[4], torna-se indispensável a preocupação do desenvolvedor em otimizar o tempo de execução de seus algoritmos.

Existem diversas implementações para a linguagem, o que abre possibilidade para a variação da complexidade a depender do nível de otimização de cada *engine.* Além disso, mesmo de posse do código-fonte, devido à natureza altamente complexa da implementação de uma linguagem de programação, não é trivial extraírmos esta informação pela simples análise visual do código.

Neste trabalho, buscamos entender a relação de complexidade de tempo de alguns métodos nativos dos objetos *Array* e *String* do JavaScript versus implementações conhecidas para a resolução dos mesmos problemas. Desta forma, conseguimos gerar gráficos de tempo de execução médio e determinar com mais precisão o comportamento assintótico das funções, bem como quais abordagens se mostraram mais eficientes.

2. Complexidade de tempo em algoritmos

A complexidade de tempo de um algoritmo pode ser definida como uma propriedade deste que indica, de forma abstrata, a quantidade de tempo que o código levará para ser executado de acordo com o tamanho da sua entrada. Esta propriedade, que assume a forma de uma fórmula matemática, é calculada considerando-se que as operações básicas[1] são executadas em tempo constante, e, portanto, busca-se uma equação que calcule a quantidade de execuções de operações básicas dado o tamanho de entrada N.

Como a complexidade de tempo de um algoritmo pode variar bastante de acordo com o tamanho da entrada e/ou o estado dos dados fornecidos, convenciona-se que uma abordagem eficiente de cálculo de complexidade é utilizando-se do conceito de "complexidade de pior caso". Nesta abordagem, considera-se que a complexidade do algoritmo é dada pela complexidade obtida no pior cenário dos dados - ou seja, aquela configuração dos dados de entrada que gera a maior quantidade de operações executadas possível.

A complexidade de pior caso é uma forma eficiente e conservadora de se analisar a complexidade de um algoritmo, mas não é capaz de prever o custo de todas as execuções realizadas com entradas diferentes mas de mesmo tamanho. Como esse cálculo preciso de quantidade de operações básicas executadas para um dado *dataset* é pouco prático em nosso dia-a-dia, comumente utilizamos juntamente à complexidade de pior caso a notação *Grande-O* (*Big O notation)*. Essa notação, representada pela letra "O" seguida de parênteses englobando uma função, indica uma limitação assintótica superior da função. Em outras palavras, ela indica que a complexidade do algoritmo vai atingir no máximo o valor daquela função para um dado tamanho de entrada N, nunca acima.

[1] *Operações básicas incluem, mas não se restringem à, operações de atribuição de valor a variáveis, soma, subtração, multiplicação, divisão, mod (resto de divisão), shift de bits e comparação.*

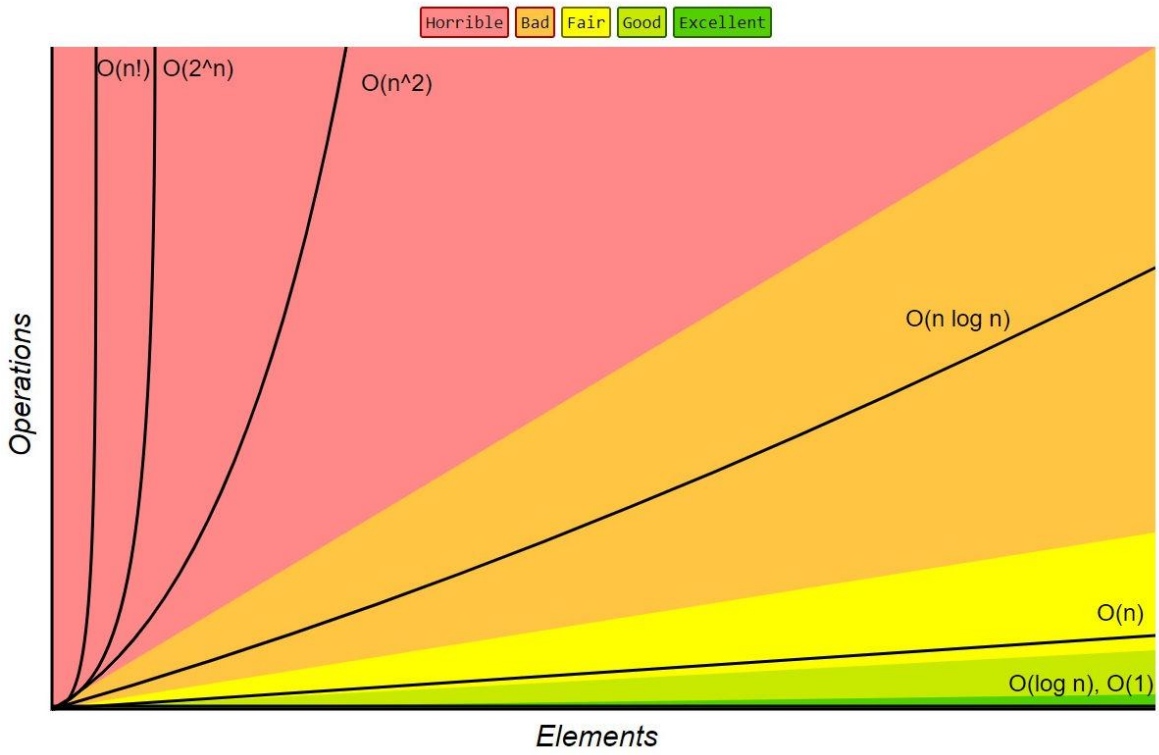

Figura 1: diferentes complexidades de algoritmos representadas na notação *Big-O*.[5]

A Figura 1 traz diferentes complexidades de algoritmos representadas na notação *Big-O* dentro de uma escala arbitrária, mas largamente utilizada dentro da comunidade, de eficiência do código, que vai de "Horrível" (*Horrible*) à "Excelente" (*Excellent*). Nela podemos ver claramente o problema de algoritmos de alta complexidade: o comportamento assintótico dessas funções atinge valores de operações (e consequentemente, segundos) próximos ao infinito muito mais rapidamente com o crescimento de N (elementos) do que funções de menor complexidade, tornando o processamento de grandes quantidades de dados inviável ou extremamente caro e demorado. Por outro lado, algoritmos de baixa complexidade, como *O(log n) ou O(n)*, conseguem lidar com quantidades muito maiores de dados antes de se tornarem inviáveis.

3. JavaScript

O JavaScript é uma linguagem de programação classificada como leve, interpretada e baseada em objetos com funções de primeira classe. Foi inicialmente criada como uma linguagem de script para navegadores pela ***Netscape*** em 1995, que desejava permitir que os sites da internet tivessem comportamentos dinâmicos, e não somente estáticos. Após anos de disputa com o ***JScript***, linguagem de script de mesmo propósito criada pela Microsoft para rodar em seu navegador, o Internet Explorer, o JavaScript eventualmente se consolidou como a linguagem padrão a ser utilizada nos navegadores modernos.

Deu origem e é baseada no ECMAScript, uma linguagem de script padronizada pela ***Ecma International*** através das especificações ECMA-262 e ECMA-402. Esta padronização busca coordenar as implementações de interpretadores da linguagem, permitindo uma grande compatibilidade entre diferentes navegadores, o que não acontecia na época da disputa entre **JScript** e **JavaScript**.

Sua adoção ganhou proporções ainda maiores em 2009, com a criação do **Node.js**, um ambiente de execução independente de browsers baseado no interpretador **V8** do Google Chrome. Ao adicionar um loop de eventos e API's de I/O, Ryan Dahl, o criador do Node.js, abriu portas para usos mais complexos da linguagem, inclusive em projetos back-end.

Hoje, o JavaScript está presente em 98% dos sites da internet, e o ***npm***, um popular gerenciador de pacotes da linguagem, é o gerenciador com mais módulos do mundo.

3.1. Principais interpretadores

Interpretadores JavaScript são softwares que executam código JavaScript. Inicialmente, todos os interpretadores simplesmente executavam o código JavaScript, sem nenhum tipo de compilação realizada previamente. Isso mudou em 2008, com o lançamento do Google Chrome e seu interpretador, o V8. Este novo interpretador foi o primeiro a implementar o ***just-in-time compilation (JIT)***. Este método de compilação

compila o código durante a execução, e não antes, como é de costume, o que permite que o compilador identifique e otimize apenas trechos de código que se beneficiam da compilação.

O lançamento do V8 revolucionou o mundo dos browsers, e forçou seus concorrentes a desenvolverem seus próprios interpretadores com **JIT.** Hoje, todo grande navegador possui sua versão de interpretador com **JIT**. Dentre os principais, podemos citar:

- **V8:** interpretador utilizado pelo Google Chrome. Também é base para o Node.js.
- **SpiderMonkey:** é o interpretador desenvolvido pela Mozilla, utilizado no navegador Firefox.
- **JavaScriptCore:** é o interpretador desenvolvido pela Apple para ser utilizado no Safari, navegador da empresa.
- **Chakra:** foi o interpretador utilizado no Internet Explorer e no início do Edge, ambos navegadores da Microsoft. O Edge eventualmente foi reconstruído utilizando como base o **V8**.

4. Complexidade de funções nativas do JavaScript

Para esta pesquisa, definimos como objetivo principal comparar o desempenho de execução de algumas funções nativas do JavaScript versus implementações comuns de algoritmos para resolver o mesmo problema.

Para medir o desempenho e evitar eventuais inconsistências nos dados coletados, optamos pela utilização do ***benchmark.js***, uma biblioteca JavaScript especializada em medir o desempenho de algoritmos.

4.1. Funções analisadas

As funções analisadas nesta pesquisa foram escolhidas dentro do grupo dos métodos nativos dos objetos ***Array*** e ***String***, presentes em todas as implementações do

JavaScript. O critério de escolha baseou-se no nível de complexidade da operação realizada por estas (quanto mais complexo, melhor) e a existência de algoritmos conhecidos para resolver o mesmo problema.

Abaixo, listamos as funções analisadas para cada tipo de objeto:

Array: *concat, find, reverse, shift, unshift, sort e push.*

String*: match, substring e toUpperCase.*

4.2. Datasets

Para a realização dos testes, é primordial que tenhamos uma vasta gama de dados - compatíveis com as funções testadas - disponível. Por isso, desenvolvemos um algoritmo responsável por criar datasets de ***Arrays*** e ***Strings*** de tamanhos e formatos variados.

Cada função foi então testada em cima de um ou mais datasets diferentes, para testar seu comportamento em situações variadas.

Para *Arrays*, geramos *datasets* contendo *Arrays* com 2 a 100.000 elementos organizados como números em ordem decrescente, números negativos e números randômicos. Para *Strings*, geramos *datasets* de palavras e frases aleatórios com quantidade de caracteres variando entre 2 e 100.000 itens.

4.3. A biblioteca (*benchmark.js*)

O ***benchmark.js*** é uma biblioteca JavaScript voltada para a realização de testes de performance de algoritmos. Através da utilização de ***high resolution timers***[6], a biblioteca é capaz de medir o tempo de execução de uma função com uma precisão de menos de um milissegundo[7], e usa este dado para fornecer dados estatísticos relevantes sobre a medição, como o tempo médio de execução e sua margem de erro.

De forma a minimizar a incerteza nas medições, definimos os parâmetros do teste para que este executasse cada função em torno de 90 vezes para cada dataset. Depois, exportamos os resultados de cada teste para arquivos *.csv* de forma a facilitar a manipulação dos dados.

A análise da biblioteca fornece diversos dados, sendo eles:

- ***mean:*** média aritmética dos tempos de execução, em segundos
- ***moe:*** margem de erro do ***mean****,* em segundos
- ***rme:*** margem de erro expressa como porcentagem do ***mean***

Para as análises gráficas, optamos por utilizar apenas o parâmetro ***mean***, que retorna o tempo de execução médio. O uso da margem de erro (***moe***) foi considerado, mas devido à alta quantidade de repetições que definimos nos parâmetros do teste, os erros se mostraram muito pequenos para fornecer dados relevantes para a análise em questão.

5. Análises para Array

5.1. Array.concat()

O método *Array.concat()* cria e retorna um novo objeto Array, composto pelos elementos que compõem o *Array* a partir do qual o método foi executado junto dos elementos de todos os *Arrays* passados como parâmetro da função[8]. A sintaxe do método é apresentada no Algoritmo 1.

```
arr.concat(valor1, valor2, ..., valorN)
```

Algoritmo 1: sintaxe do método *Array.concat()* no JavaScript[8].

Ou seja, para um dado *array1* e um *array2*, ao executarmos o método *concat()* no primeiro passando o segundo como parâmetro, teremos como resultado um novo Array contendo os elementos de ambos, como mostrado no Algoritmo 2 abaixo.

```
> let array1 = [1, 4, 5];
> let array2 = [2, 3, 4];
```

```
> array1.concat(array2);
< [1, 4, 5, 2, 3, 4] //output
```

Algoritmo 2: exemplo de uso do método *concat()* com dois *Arrays*

Para a análise comparativa deste método, implementamos dois algoritmos diferentes.

Na primeira abordagem, utilizamos dois *for loops* para popular um novo *Array* previamente instanciado com o tamanho correto do *Array* de saída. Este algoritmo resulta em uma complexidade *O(n+m)*, onde n e m são os comprimentos dos arrays, como podemos ver no Algoritmo 3 abaixo.

```
function concat1(arr1, arr2) {
    const length = arr1.length + arr2.length;
    const resultsArray = new Array(length);
    let i, index;

    for(i = 0; i < arr1.length; i++){              O(n)
        resultsArray[i] = arr1[i];
    }
    for(i = arr1.length; i < length; i++){         O(m)
        index = i - arr1.length;                     O(1)
        resultsArray[i] = arr2[index];               O(1)
    }
    return resultsArray;                           Total: O(m+n)
}
```

Algoritmo 3: implementação de um algoritmo equivalente ao *concat()* utilizado em nossos testes e sua respectiva análise de complexidade.

Na segunda abordagem, optamos por um algoritmo levemente diferente do original - neste, alteramos o *Array* original, a fim de entendermos se desta forma teríamos um melhor desempenho do que a função nativa. Caso positivo, em casos onde o *Array* original não fosse necessário, poderíamos recorrer a este algoritmo para uma melhor performance.

A complexidade calculada para este algoritmo foi de *O(m)*, como podemos ver no Algoritmo 4 abaixo.

```
function concat2(arr1,arr2) {
    let newLength = arr1.length + arr2.length;        O(1)
    let index = 0;                                    O(1)
    let i;                                            O(1)

    for(i = arr1.length; i < newLength; i++) {        O(m)
       arr1[i] = arr2[index];                           O(1)
       index++;                                         O(1)
    }
}                                                     Total: O(m)
```

Algoritmo 4: segunda implementação de um algoritmo equivalente ao *concat()* utilizado em nossos testes e sua respectiva análise de complexidade

Na Figura 2 abaixo podemos ver o gráfico comparando o desempenho das 3 abordagens quando aplicadas nos *datasets* de números aleatórios.

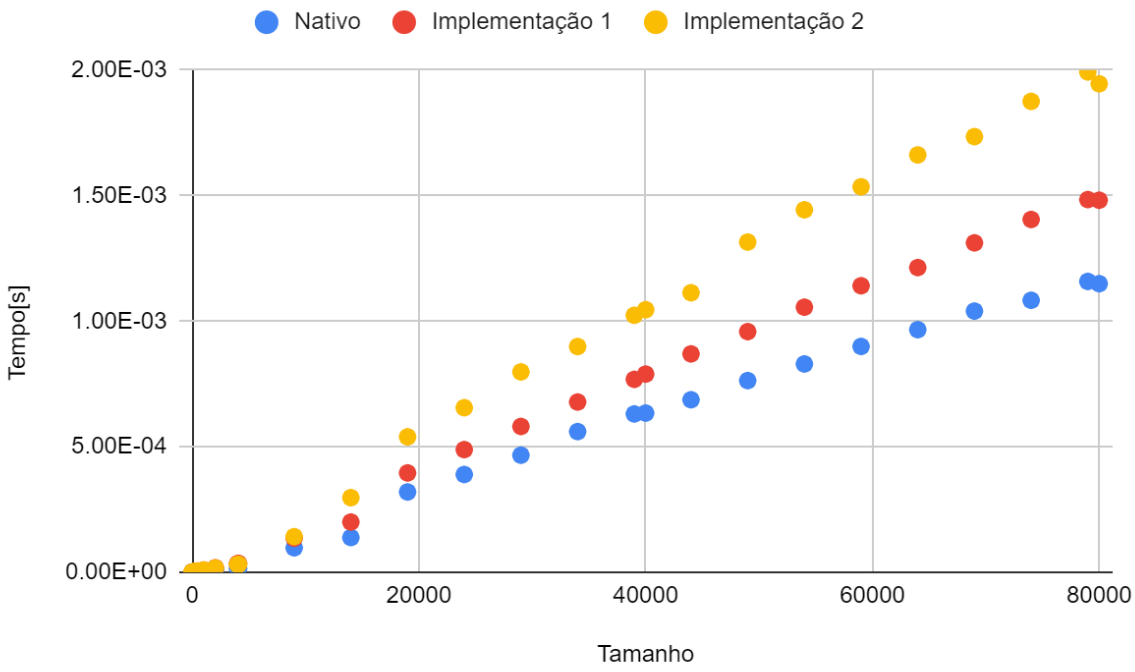

Figura 2: Gráfico comparando o tempo médio de execução das implementações da função concat() testadas sobre o *dataset* de números aleatórios.

A partir da análise do gráfico, podemos concluir que:

- o método nativo se mostrou mais eficiente do que o *concat1* e o *concat2;*
- embora levemente diferentes, todos os algoritmos mostraram um comportamento assintótico similar, o que **nos leva a crer que o algoritmo nativo possui complexidade O(n+m);**
- o *concat2*, embora teoricamente mais eficiente (com complexidade O(m)), se mostrou o menos eficiente de todos. Uma possível explicação para essa perda de eficiência pode ser o fato de alocarmos mais espaço na memória iterativamente para cada novo item, ao invés de alocarmos tudo de uma vez no início. Movidos por este resultado, testamos a complexidade desta operação mais adiante no trabalho (método *Array.push()*), e descobrimos que sua complexidade de pior caso efetiva é de *O(n)*, o que explica o pior resultado deste algoritmo.

5.2. Array.find()

O método *Array.find()* recebe como parâmetro uma função de teste e retorna o valor do primeiro elemento que satisfizer a função. Caso não exista nenhum valor que a satisfaça, o método retorna *undefined*[9]. A sintaxe do método é apresentada no Algoritmo 5.

```
arr.find(callback(element[, index[, array]])[, thisArg])
```

Algoritmo 5: sintaxe do método *Array.find()* no JavaScript.[9]

Para exemplificar o funcionamento deste método, podemos usar como exemplo um *Array*, *array1*, com alguns elementos inteiros, e uma função de teste, *testFunction*, que procura por elementos maiores do que 10. Um exemplo de execução é apresentado no Algoritmo 6.

```
> let array1 = [1, 2, 5, 12];
> let testFunction = (element) => element > 10;
```

```
> array1.find(testFunction);
< 12 //output
```

Algoritmo 6: exemplo de uso do método Array.find().

Para a análise de desempenho deste método, implementamos uma segunda abordagem de complexidade conhecida.

A complexidade da implementação do *find()* depende de dois fatores, o tamanho do *Array* e a função *fn* aplicada. Como utilizamos a mesma função *fn* para ambos algoritmos, podemos desconsiderar o acréscimo de complexidade gerado por ela, pois é o mesmo em ambos os casos. Portanto, para o algoritmo *find()* implementado, temos a complexidade dada pelo tamanho do *Array* e o custo de execução da função utilizada, que foi de *O(1)*.

Podemos ver o algoritmo e sua análise de complexidade no Algoritmo 7.

```
function find(array, fn) {
    for(let i = 0; i < array.length; i++) {          O(n)
        if (fn(array[i])) {                            O(1)
            return array[i];
        }
    }
    return undefined;
}                                                    Total: O(n)
```

Algoritmo 7: implementação de um algoritmo equivalente ao *Array.find()* utilizado em nossos testes e sua respectiva análise de complexidade.

Na Figura 3 podemos ver o gráfico comparando o desempenho das 2 abordagens quando executadas em cima do *dataset* de números aleatórios.

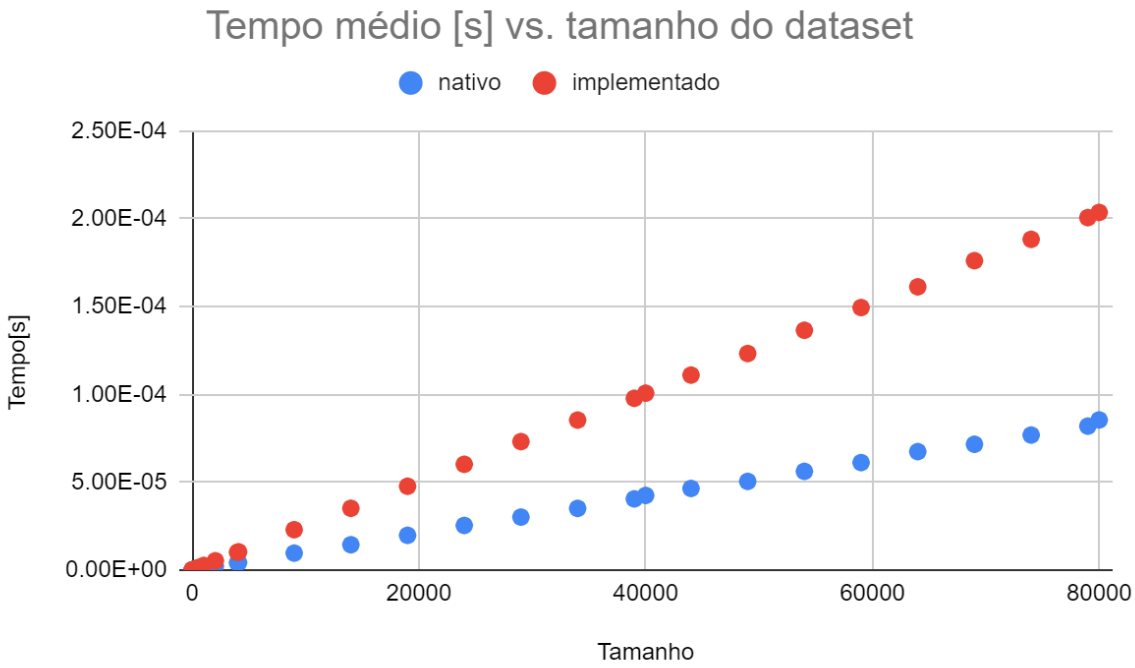

Figura 3: Gráfico comparando o tempo médio de execução das implementações da função *find()* testadas sobre o *dataset* de números aleatórios.

Resultados similares foram encontrados no tempo de execução de ambos algoritmos para os *datasets* de números decrescentes e negativos. A partir da análise do gráfico, podemos concluir que:

- o método nativo possui uma complexidade de tempo consideravelmente menor do que o implementado;
- embora diferentes, ambos possuem uma taxa de crescimento linear, compatível com a complexidade *O(n)* do algoritmo implementado;
- conclui-se portanto que, em todas as situações, é sempre mais vantajoso utilizar o método nativo, pois este apresenta otimizações que o tornam mais eficiente.

5.3. Array.reverse()

O método *Array.reverse()*, quando executado em um *Array*, inverte a posição de todos os seus elementos, transformando o primeiro elemento em último e vice-versa, e assim sucessivamente[10]. A sintaxe do método é apresentada no Algoritmo 8.

```
arr.reverse()
```

Algoritmo 8: sintaxe do método *Array.reverse()*.[10]

Para exemplificar seu funcionamento, apresentamos um exemplo de seu uso no Algoritmo 9.

```
> let array = [1, 2, 3, 4];
> array.reverse();

< [4, 3, 2, 1] //output
```

Algoritmo 9: exemplo de uso do método *Array.reverse()*.

Existem diversas formas de inverter a ordem dos elementos de um algoritmo e, portanto, para este teste implementamos três algoritmos diferentes para comparação.

O primeiro algoritmo, apresentado no Algoritmo 10, cria um novo *Array* vazio, *rev,* de tamanho zero. Em seguida, percorre o *array* a ser invertido de trás para frente, adicionando os elementos um a um em *rev.*

```
function customReverse(input) {
  var rev = [];                                  O(1)
  for(let i = input.length-1; i >= 0; i--) {     O(n)
      rev.push(input[i]);                          O(n)
  }
  return rev;                                    Total: O(n²)
}
```

Algoritmo 10: primeira implementação de um algoritmo equivalente ao *Array.reverse()* utilizado em nossos testes e sua respectiva análise de complexidade.

O segundo algoritmo, apresentado no Algoritmo 11, inverte os elementos do *Array in-place* (dentro do próprio *Array*, sem utilizar-se de um *Array* secundário). Nesta abordagem utilizamos um operador específico do JavaScript que permite alternarmos o valor de duas variáveis sem a utilização de uma variável auxiliar explícita.

```
function customReverse2(input) {
  let init = 0;                                  O(1)
  let end = input.length - 1;                    O(1)
  while (init < end) {                           O(n/2) = O(n)
    [input[init], input[end]] =                    O(1)
      [input[end], input[init]];
    init++;                                        O(1)
    end--;                                         O(1)
  }
  return input;                                  Total: O(n)
}
```

Algoritmo 11: segunda implementação de um algoritmo equivalente ao *Array.reverse()* utilizado em nossos testes e sua respectiva análise de complexidade

Por fim, o terceiro algoritmo segue uma abordagem similar ao segundo algoritmo, mas utiliza uma variável auxiliar ao invés da sintaxe específica do JavaScript, como podemos ver no Algoritmo 12.

```
function customReverse3(input) {
  let init = 0;                                  O(1)
  let end = input.length - 1;                    O(1)
  while (init < end) {                           O(n/2) = O(n)
    const aux = input[init];                       O(1)
    input[init] = input[end];                      O(1)
    input[end] = input[aux];                       O(1)
    init++;                                        O(1)
    end--;                                         O(1)
  }
  return input;                                  Total: O(n)
}
```

Algoritmo 12: terceira implementação de um algoritmo equivalente ao *Array.reverse()* utilizado em nossos testes e sua respectiva análise de complexidade

Na Figura 4 podemos ver o gráfico comparando o desempenho das 4 abordagens quando executadas com o *dataset* de números aleatórios como entrada.

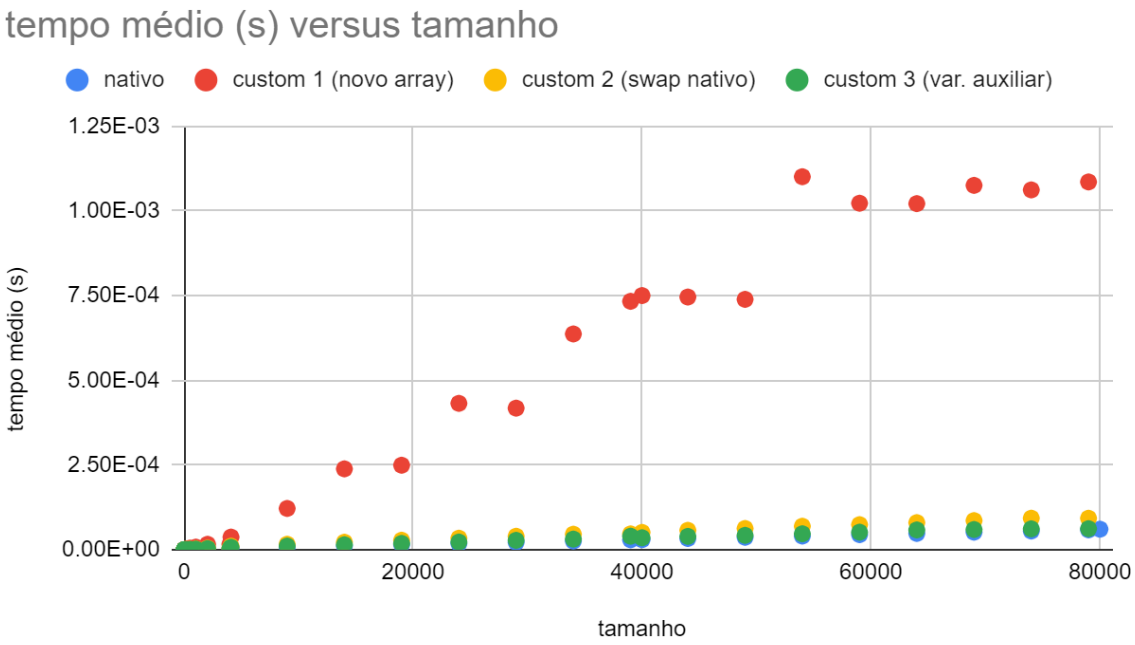

Figura 4: Gráfico comparando o tempo médio de execução das implementações da função *reverse()* testadas sobre o *dataset* de números aleatórios.

Como podemos observar, o tempo médio de execução da primeira abordagem implementada foi diversas escalas de grandeza maior do que as outras 3 abordagens. Este comportamento era esperado, já que a complexidade calculada deste algoritmo foi de $O(n^2)$, enquanto que a complexidade dos outros foi de $O(n)$. Porém, embora esperado, este resultado exemplifica bem a gigantesca disparidade de tempo necessário para executar métodos mais complexos versus seus equivalentes mais simples, e a necessidade de nos atentarmos à complexidade dos nossos algoritmos.

Para podermos fazer uma comparação melhor dos métodos menos complexos, ocultamos grande parte dos pontos da curva da primeira abordagem, como pode ser visto na Figura 5.

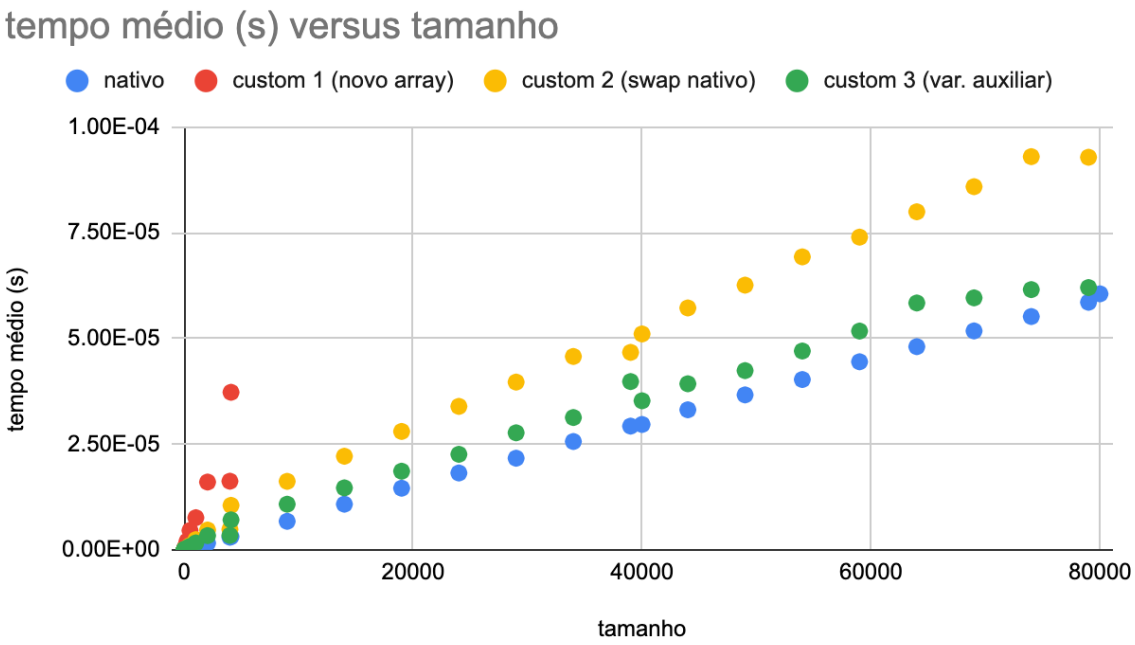

Figura 5: Gráfico comparando o tempo médio de execução das implementações da função *reverse()* testadas sobre o *dataset* de números aleatórios. Diversos pontos da curva do algoritmo *custom 1* foram ocultadas para permitir a melhor visualização da curva dos outros algoritmos.

Assim como no *find()*, resultados similares foram observados quando executamos os algoritmos nos *datasets* de números decrescentes e negativos. Assim, podemos tirar as seguintes conclusões a partir da análise desses gráficos:

- a diferença de tempo necessário para a execução de algoritmos $O(n^2)$ e $O(n)$ é muito grande, e portanto a preocupação com a complexidade dos nossos algoritmos é muito importante;
- os comportamentos assintóticos das funções $O(n)$ se mostraram muito similares, com uma pequena vantagem para o método nativo;
- o método nativo, por sua vez, possui complexidade $O(n)$;
- a estrutura específica para *swap* de valores entre variáveis do JavaScript se mostrou menos eficiente do que o uso de uma variável auxiliar, já que a segunda abordagem apresentou uma curva mais acentuada do que a terceira.

5.4. Array.shift()

O método *Array.shift()* remove o primeiro elemento do *Array* e retorna seu valor. Nesta operação, o *Array* original é modificado, bem como seu comprimento[11]. A sintaxe da função é apresentada no Algoritmo 13.

```
arr.shift()
```

Algoritmo 13: sintaxe do método Array.shift().[11]

Para exemplificar seu uso, apresentamos um exemplo do seu funcionamento no Algoritmo 14.

```
> const array = [1, 2, 3, 4];
> array.shift();
< 1 //output
> console.log(array);
< [2, 3, 4] //output
```

Algoritmo 14: exemplo de uso do método Array.shift().

Como função de comparação, implementamos um algoritmo equivalente que move os elementos do *Array* uma casa para frente e, ao final, reduz seu comprimento em 1, deletando o último elemento (que já havia sido movido para a casa imediatamente anterior). O algoritmo é apresentado no Algoritmo 15.

```
function shift(array){
    const newlength = array.length-1;          O(1)
    for(i=0; i < newlength; i++){              O(n)
        array[i] = array[i+1];                   O(1)
    }
    array.length = newlength;                  O(1)
}                                              Total: O(n)
```

Algoritmo 15: implementação de um algoritmo equivalente ao *Array.shift()* utilizado em nossos testes e sua respectiva análise de complexidade.

Na Figura 6 podemos ver o gráfico comparando o desempenho das 2 abordagens

quando executadas em cima do *dataset* de números aleatórios.

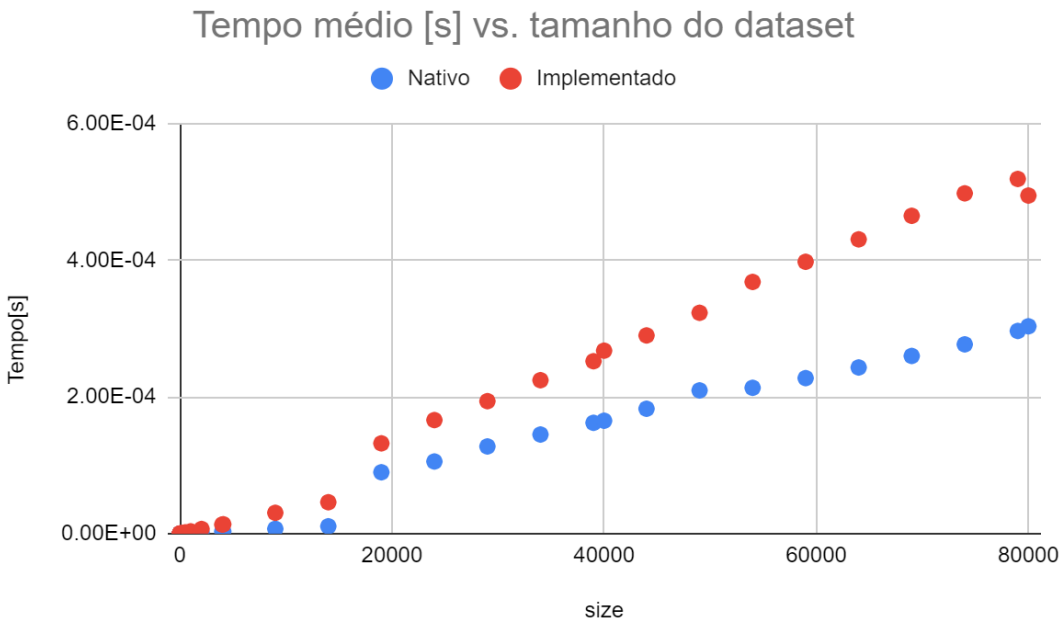

Figura 6: Gráfico comparando o tempo médio de execução das implementações da função *shift()* testadas sobre o *dataset* de números aleatórios.

A partir da análise do gráfico, podemos observar que:

- a função manualmente implementada se mostrou menos eficiente para todos os tamanhos de dataset testados;
- embora menos acentuada, a inclinação da curva de tempo médio do método nativo a mantém constantemente abaixo da função *O(n)*, o que nos permite afirmar que a função nativa possui complexidade *O(n)*.

5.5. Array.unshift()

O método *Array.unshift()* recebe N parâmetros e os adiciona, na ordem recebida, ao início do *Array*, retornando o tamanho final do objeto ao final da execução[12]. A sintaxe do método é apresentada no Algoritmo 16.

```
arr.unshift([element1[, ...[, elementN]]])
```

Algoritmo 16: sintaxe do método Array.unshift().[12]

O Algoritmo 17 exemplifica seu funcionamento.

```
> const array = [1, 2, 3, 4];
> array.unshift(0);
< 5 //output
> console.log(array);
< [0, 1, 2, 3, 4] //output
```

Algoritmo 17: exemplo de uso do método Array.unshift().

Para análise de comparação, implementamos um algoritmo equivalente que adiciona um elemento ao início de um *Array*. O algoritmo percorre o *Array* recebido como parâmetro de trás para frente, movendo todos os seus itens uma casa para frente e, ao final, insere o elemento recebido na primeira casa. O algoritmo é apresentado no Algoritmo 18.

```
function unshift(array, element) {
    for (let i = array.length; i > 0; i--) {    O(n)
        array[i] = array[i-1];                    O(1)
    }
   array[0] = element;                           O(1)
}
```

Algoritmo 18: implementação de um algoritmo equivalente ao *Array.unshift()* utilizado em nossos testes e sua respectiva análise de complexidade.

Na Figura 7 podemos ver o gráfico comparando o desempenho dos dois algoritmos quando aplicado em cima do *dataset* de números aleatórios.

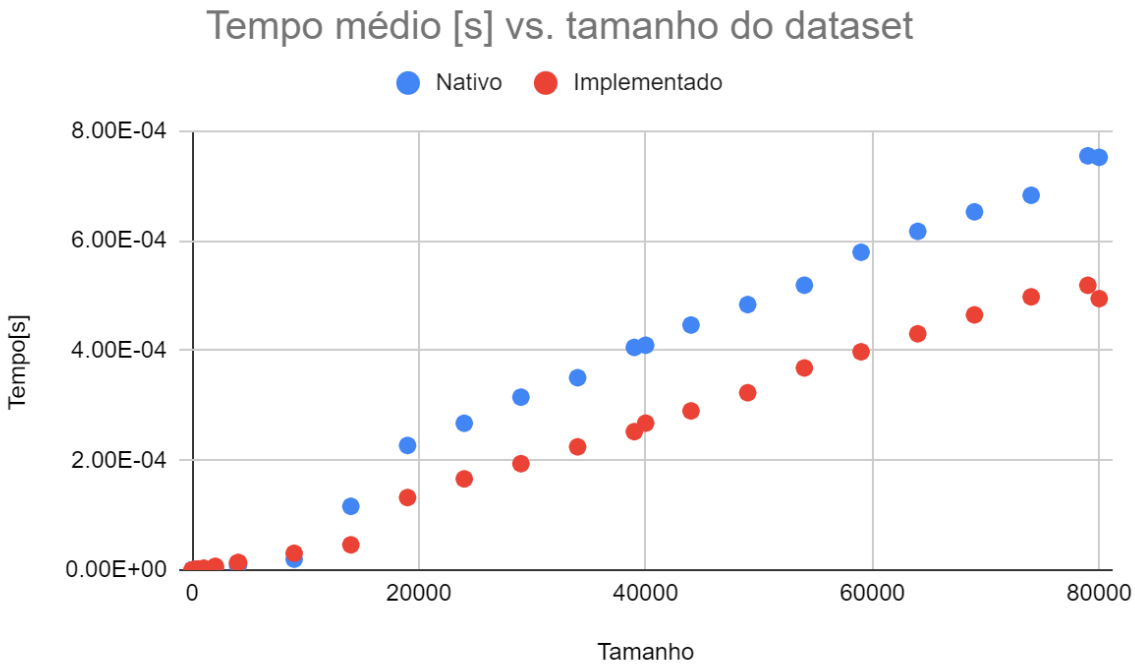

Figura 7: Gráfico comparando o tempo médio de execução das implementações da função *shift()* testadas sobre o *dataset* de números aleatórios.

A partir da análise do gráfico, podemos concluir que:

- os algoritmos apresentam comportamentos assintóticos similares, com uma pequena vantagem no tempo de execução para o algoritmo implementado;
- Embora levemente mais demorado, o método nativo ainda pode ser considerado um algoritmo *O(n)*, pois constantes são desconsideradas nessa notação.

5.6. Array.sort()

O método *Array.sort()* ordena os elementos do *Array*. A ordenação padrão, realizada quando não é fornecida uma função de comparação como parâmetro, realiza uma ordenação crescente baseada na pontuação de código *unicode*. Caso seja fornecida uma função de comparação, esta deve receber 2 parâmetros, x e y, e compará-los segundo os critérios desejados. Caso x deva vir antes de y, o retorno da função deve ser menor do que

0. Para que y venha antes de y, a função deve retornar um valor maior do que 0. Caso ela retorne 0, a posição relativa dos elementos é mantida inalterada, mas eles continuam sendo ordenados com relação aos outros elementos do Array[13]. A sintaxe do método é apresentada no Algoritmo 19.

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

Algoritmo 19: sintaxe do método Array.sort().[13]

Para demonstrar seu uso, apresentamos um exemplo do seu funcionamento no Algoritmo 20.

```
> const array = [9, 5, 3, 1];
> array.sort();
< [1, 3, 5, 9] //output
```

Algoritmo 20: exemplo de uso do método Array.sort().

Existem diversos algoritmos conhecidos para a ordenação de um *Array*, com diversas complexidades diferentes e casos de uso específico. Para a realização deste teste, optamos por utilizar o conjunto de *datasets* ordenados em ordem decrescente, pois esta configuração costuma ser o cenário de pior caso para os algoritmos de ordenação.

Para a comparação, implementamos 3 algoritmos conhecidos de ordenação, o *Selection Sort,* o *Quick Sort* e o *Merge Sort*.

O *Selection Sort,* apresentado no Algoritmo 21, é um algoritmo de ordenação de complexidade $O(n^2)$. Ele percorre o *Array* comparando cada elemento com todos os outros que o sucedem, na busca do menor elemento. Ao final da comparação, ele troca o menor elemento encontrado no *SubArray* com o elemento inicial, e reinicia a busca pelo menor elemento partindo da próxima casa, e assim sucessivamente.

```
function selectionSort(inputArr) {
    let n = inputArr.length;                    O(1)

    for(let i = 0; i < n; i++) {                O(n)
        let min = i;                              O(1)
        for(let j = i+1; j < n; j++){             O(n - 1)
```

```
            if(inputArr[j] < inputArr[min]) {       O(1)
                min=j;                              O(1)
            }
        }                                           O(1)
        if (min != i) {                             O(1)
            let tmp = inputArr[i];                  O(1)
            inputArr[i] = inputArr[min];            O(1)
            inputArr[min] = tmp;
        }                                           Total: O(n²)
    }
    return inputArr;
}
```

Algoritmo 21: implementação do *Selection Sort* utilizado em nossos testes e sua respectiva análise de complexidade.

O *Quick Sort*, apresentado no Algoritmo 22, é um algoritmo de ordenação também de complexidade de pior caso $O(n^2)$. Embora esta afirmação passe a impressão de que este é um algoritmo de ordenação ineficiente, o *Quick Sort* é geralmente mais rápido na prática do que outros algoritmos de complexidade $O(n.log(n))$, como o *Merge Sort*, pois realiza operações *in-place*, diminuindo drasticamente a necessidade de uso de memória auxiliar. Além disso, podemos reduzir o risco de cairmos no pior caso ao utilizarmos pivôs randômicos. Com tudo isso, fazendo uma análise mais realista, podemos afirmar que o *Quick Sort* possui complexidade de tempo média $O(n.log(n))$. A implementação que utilizamos para testar o tempo médio de execução do *Quick Sort* é apresentada no Algoritmo 22.

```
function swap(items, leftIndex, rightIndex){
    let temp = items[leftIndex];                O(1)
    items[leftIndex] = items[rightIndex];       O(1)
    items[rightIndex] = temp;                   O(1)
}                                               Total: O(1)
function partition(items, left, right) {
    let pivot = items[Math.floor((right +
left) / 2)];                                    O(1)
    let i = left;                               O(1)
    let j = right;                              O(1)
```

```
    while (i <= j) {                                O(n)
        while (items[i] < pivot) {                   O(n)
            i++;                                      O(1)
        }
        while (items[j] > pivot) {                   O(n)
            j--;                                      O(1)
        }
        if (i <= j) {                                O(1)
            swap(items, i, j);                        O(1)
            i++;                                      O(1)
            j--;                                      O(1)
        }
    }
    return i;                                       O(1)
}                                                   Total: O(n²)

function quickSort(items, left, right) {
    let index;                                      O(1)
    if (items.length > 1) {                         O(1)
        index = partition(items, left, right);       O(n²)
        if (left < index - 1) {                      O(1)
            quickSort(items, left, index - 1);        T(n/2)
        }
        if (index < right) {                         O(1)
            quickSort(items, index, right);           T(n/2)
        }
    }
    return items;                                   O(1)
}                                                   Total: O(n²)
```

Algoritmo 22: implementação do *Quick Sort* utilizado em nossos testes e sua respectiva análise de complexidade.

Por fim, o terceiro algoritmo testado foi o *Merge Sort*, um algoritmo de ordenação de complexidade de pior caso *O(n.log(n))*. Nominalmente o algoritmo mais eficiente dentre os testados, este algoritmo tem como desvantagem a necessidade de espaço extra na escala de *O(n)*, o que diminui sua eficiência, já que a alocação, escrita e leitura de memória são

processos consideravelmente mais lentos do que o processamento de dados. Apresentamos o algoritmo do *Merge Sort* utilizado no Algoritmo 23.

```
function merge(left, right) {
  let sortedArr = [];                              O(1)

  while (left.length && right.length) {
    if (left[0] < right[0]) {                      O(1)
      sortedArr.push(left.shift());                O(n)
    } else {
      sortedArr.push(right.shift());               O(n)
    }
  }

  return [...sortedArr, ...left, ...right];        O(1)
}                                                  Total: O(n)

function mergeSort(arr) {
  const half = arr.length / 2;                     O(1)

  if (arr.length <= 1) {                           O(1)
    return arr;                                    O(1)
  }

  const left = arr.splice(0, half);                O(1)
  const right = arr;                               O(1)
  return merge(mergeSort(left), mergeSort(right)); T(n) = 2T(n/2)
}                                                  +  O(n)=
// Usando o teorema mestre obtemos                 O(n * log n)
```

Algoritmo 23: implementação do *Merge Sort* utilizado em nossos testes e sua respectiva análise de complexidade.

Na Figura 8 podemos ver o gráfico comparando o desempenho dos quatro algoritmos quando aplicado em cima do *dataset* de números ordenados em ordem decrescente.

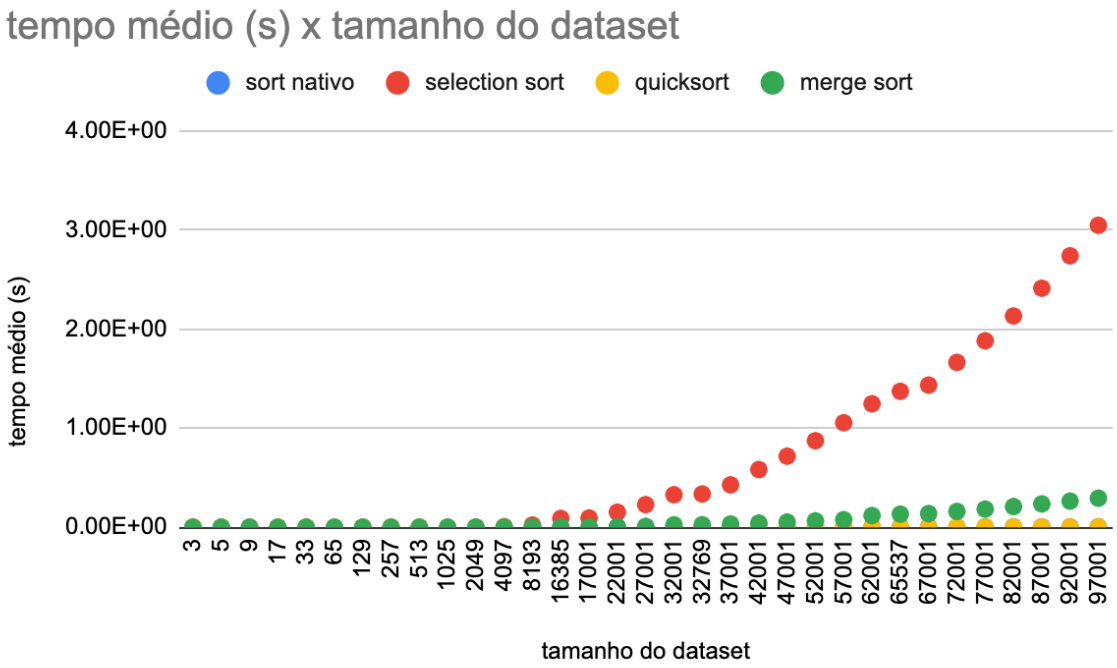

Figura 8: Gráfico comparando o tempo médio de execução das implementações da função *sort()* testadas sobre o *dataset* de números decrescentes.

A partir da análise do gráfico, fica explícito que o *Selection Sort* possui uma complexidade de tempo muito maior do que os outros, como era esperado. Esta ineficiência é ainda mais exacerbada devido à configuração do *Array*, com os elementos ordenados em ordem decrescente, que testa especificamente o pior caso do algoritmo.

Para conseguirmos analisar o desempenho dos outros algoritmos, removemos grande parte dos pontos da curva do *Selection Sort* na Figura 9.

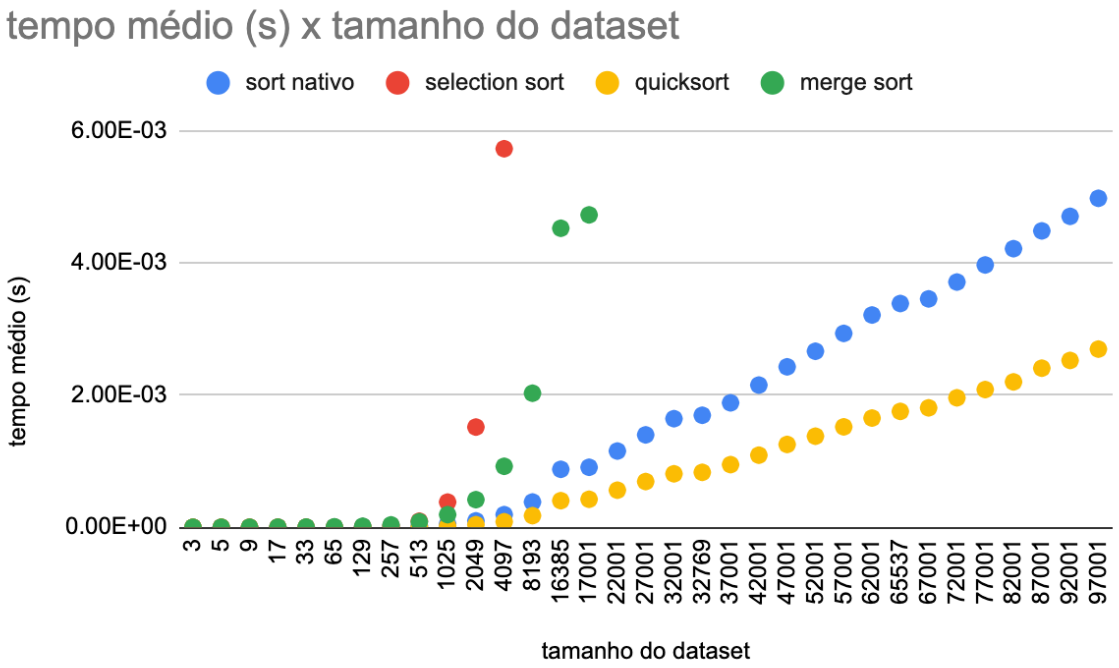

Figura 9: Gráfico ampliado comparando o tempo médio de execução das implementações da função *sort()* testadas sobre o *dataset* de números decrescentes

Neste gráfico conseguimos ver com mais clareza a diferença de desempenho entre os outros 3 algoritmos. Dentre eles, o *Merge Sort* se mostrou o menos eficiente e, surpreendentemente, o *Quick Sort* se mostrou mais eficiente do que o *sort()* nativo. O pior desempenho do *Merge Sort* pode ser justificado, como dito anteriormente, pelo seu uso exagerado de memória auxiliar comparado aos outros algoritmos. Já o desempenho do *Quick Sort* foi inicialmente inesperado, já que teoricamente, seu desempenho no pior caso é de $O(n^2)$. No entanto, é necessário lembrar que, como os arrays que utilizamos estão ordenados inversamente, a escolha do pivô é sempre ótima. Para fazer uma análise mais justa do *Quick Sort*, executamos ele e o sort nativo com o *dataset* de números aleatórios apresentado na Figura 10.

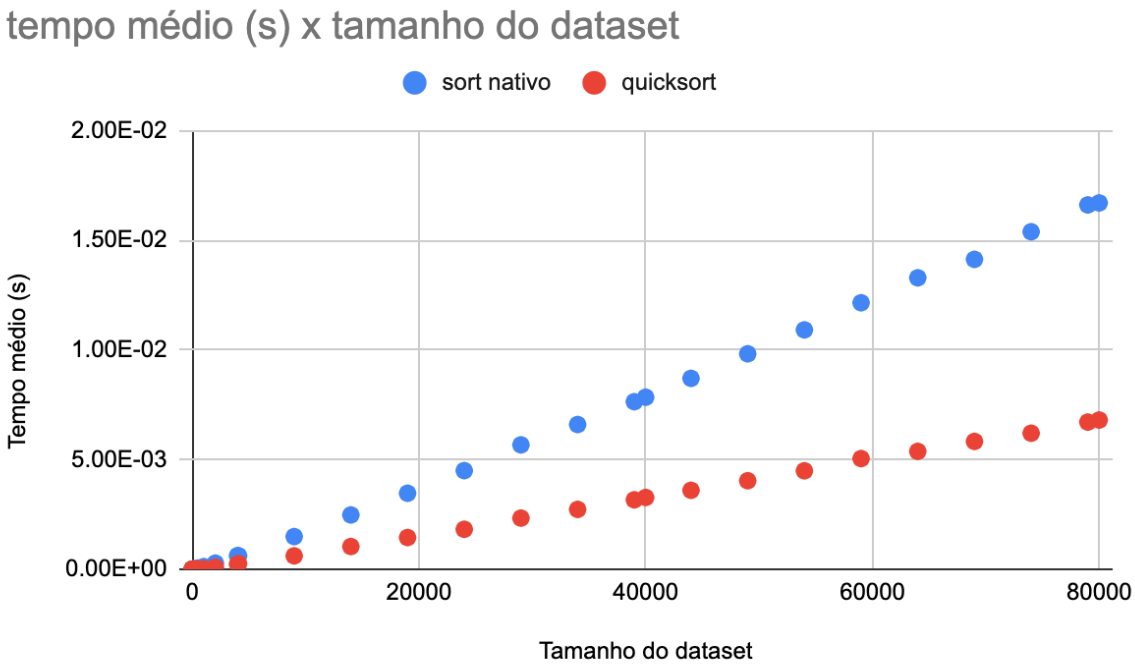

Figura 10: Gráfico comparando o tempo médio de execução das implementações da função *sort()* testadas sobre o *dataset* de números aleatórios

Como podemos observar, o *Quick Sort* continua performando melhor que o algoritmo nativo.Estes resultados mostram a importância de levarmos em consideração a alocação de memória dos algoritmos e sua complexidade de tempo médio.

Por fim, podemos concluir que a complexidade de tempo de pior caso do método nativo é de *O(n.log(n))*.

5.7. Array.push()

O método Array.push() adiciona N elementos ao final de um *Array* e retorna seu novo comprimento. O método não exige que o *Array* já tenha a quantidade de memória necessária alocada, fazendo essa alocação, quando necessária, dinamicamente[14]. A sintaxe do método é apresentada no Algoritmo 23.

```
arr.push(...items)
```

Algoritmo 23: sintaxe do método *Array.push()*.[14]

Para demonstrar seu uso, apresentamos um exemplo do seu funcionamento no Algoritmo 24.

```
> const array = [9, 5, 3, 1];
> array.push(8);
< 5 //output
> console.log(array);
< [9, 5, 3, 1, 8] //output
```

Algoritmo 24: exemplo de uso do método *Array.push()*.

Para esta análise, decidimos comparar o quão eficiente o método *push()* é quando comparado à simples adição de um novo item em um *Array* vazio mas já com o espaço correto de memória alocado. Este segundo método não substitui completamente o uso do *push()*, já que o conhecimento prévio do espaço de memória necessária para armazenar uma informação não é sempre possível, mas nos permitirá entender o quanto de desempenho estamos abrindo mão ao adotar essa abordagem.

O algoritmo de comparação é apresentado no Algoritmo 25.

```
let resultArray = new Array(array.length);
         for (let i = 0; i < array.length; i++) {
           resultArray[i] = array[i];
         }
```

Algoritmo 25: implementação de um algoritmo equivalente ao *Array.push()* utilizando um *Array* com memória pré-alocada

O algoritmo utilizado para testar o desempenho do *Array.push()* é apresentado na Figura 26.

```
let resultArray = [];
  for(let i = 0; i< array.length; i++) {
    resultArray.push(array[i]);
  }
```

Algoritmo 26: implementação do algoritmo de teste do método *Array.push()*

O gráfico comparando o tempo de execução médio das duas abordagens é apresentado na Figura 11.

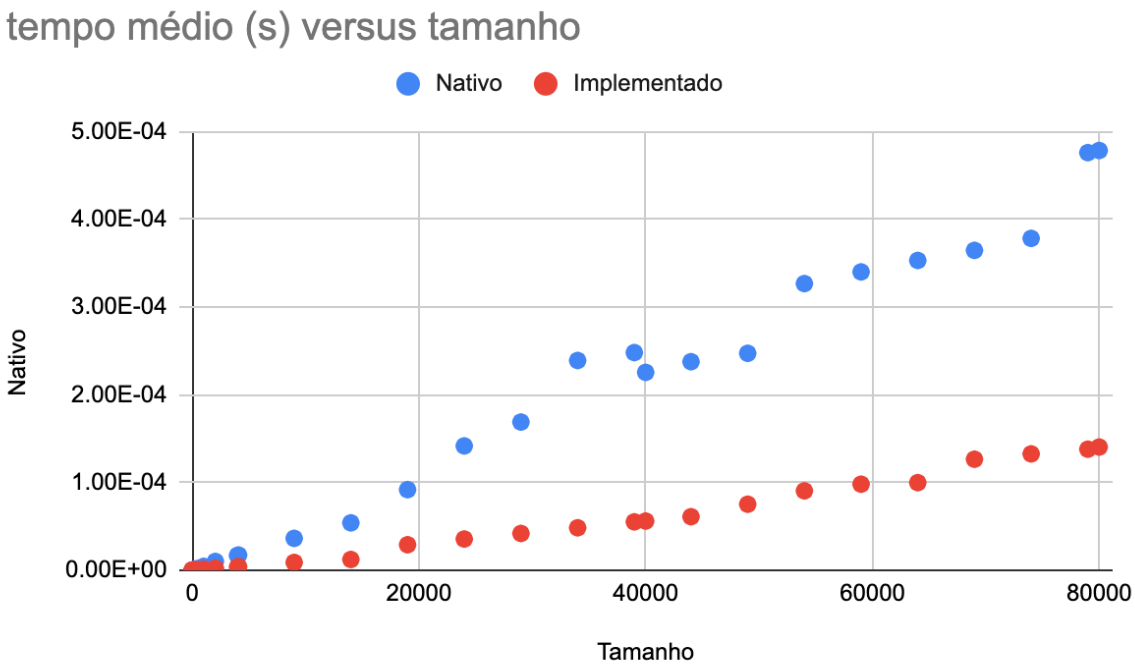

Figura 11: Gráfico comparando o tempo médio de execução das implementações da função *push()* testadas sobre o *dataset* de números aleatórios.

A partir da análise do gráfico, podemos concluir que há perda de desempenho na utilização do método *push()* ao invés da inserção de um elemento num *Array* pré-alocado, mas o desempenho se mantém na mesma ordem de grandeza, de complexidade *O(n)*. Portanto, há o comprometimento de parte do desempenho ao utilizar essa abordagem, mas não a ponto de torná-la inviável, especialmente para os casos onde não há conhecimento do tamanho final do *Array* de antemão.

6. Análises para String

6.1. String.match()

O método *String.match()* procura por uma pedaço da *String* (ou seja, uma *substring*) que satisfaça a expressão RegExp enviada como parâmetro. Caso seja encontrada, a *substring* é retornada dentro de um *Array* com algumas propriedades adicionais relacionadas à sua captura, como o *index* de início da substring. Caso não exista, o método retorna *null*[15]. A sintaxe do método é apresentada no Algoritmo 27.

```
str.match(regexp);
```

Algoritmo 27: sintaxe do método *String.match()*.[15]

Para demonstrar seu uso, apresentamos um exemplo do seu funcionamento no Algoritmo 28.

```
> let a = 'minhastring';
> a.match('nhast');
< ['nhast', index: 2, input: 'minhastring', groups: undefined]
```

Algoritmo 28: sintaxe do método *String.match()*

Como função de comparação, implementamos uma versão simplificada do método que procura por uma *substring* na *String* fornecida e retorna a posição de início no *Array*. O algoritmo utilizado, de complexidade *O(n)*, é apresentado no Algoritmo 29.

```
function match1(originalString, searchString){
 let index = 0                                          O(1)
 for (let i = 0; i < originalString.length; i++) {      O(n)
   if (originalString[i] == searchString[index]) {        O(1)
     index++;                                             O(1)
   } else {
     index = 0;                                           O(1)
   }
   if(index == searchString.length) {                   O(1)
     return i - index + 1;
   }                                                    Total: O(n)
```

```
 }
 return -1;
}
```

Algoritmo 29: implementação de um algoritmo equivalente ao *String.match()* utilizado em nossos testes e sua respectiva análise de complexidade.

O gráfico comparando o tempo de execução médio das duas abordagens é apresentado na Figura 12.

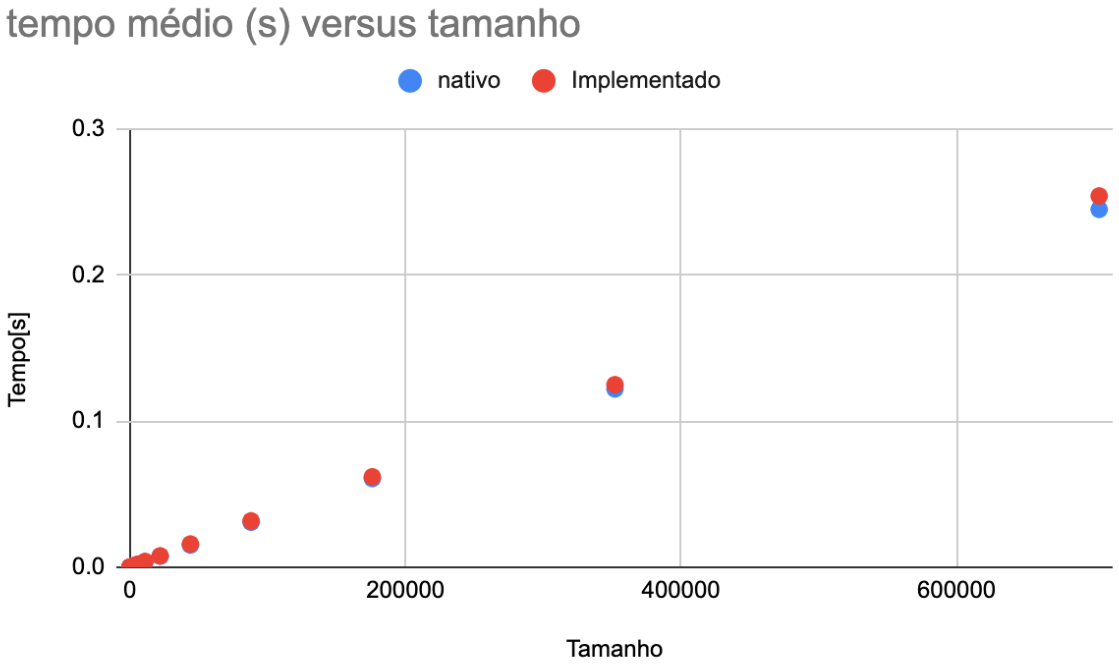

Figura 12: Gráfico comparando o tempo médio de execução das implementações da função String.*match()* testadas sobre o *dataset* de frases aleatórias.

Ambos algoritmos tiveram praticamente o mesmo desempenho, para todos os *datasets*, o que nos permite afirmar que a complexidade de tempo para o método nativo também é de *O(n)*, sendo *n* a quantidade de caracteres da *String*.

6.2. Array.substring()

O método *Array.substring()* recebe dois parâmetros, o índice inicial e o final, e retorna uma nova String correspondente à *substring* compreendida entre esses índices. O segundo parâmetro é opcional, e no caso da sua ausência considera-se que o índice final é o último índice da *String*[16]. A sintaxe do método é apresentada no Algoritmo 30.

```
str.substring(indexStart[, indexEnd])
```

Algoritmo 30: sintaxe do método *String.substring()*.[16]

Apresentamos um exemplo de uso do método no Algoritmo 31.

```
> let a = 'minhastring';
> a.substring(5, 9);
< 'stri'
```

Algoritmo 31: exemplo de uso do método *String.substring()*

Para obtermos a curva de referência para a análise de complexidade do método, implementamos um método equivalente, de complexidade de pior caso $O(n^2)$, apresentado no Algoritmo 32.

```
function implementedSubstring(string, start, end) {
 let resultString = '';                              O(1)
 for(let i = start; i <  end; i++){                  O(n)
   resultString += string[i];                          O(n)
 }
 return resultString;                                Total: O(n²)
}
```

Algoritmo 32: implementação de um algoritmo equivalente ao método *String.substring()* e sua respectiva análise de complexidade

De posse do algoritmo equivalente, realizamos o benchmark com os dois algoritmos em cima do *dataset* de frases aleatórias e apresentamos o gráfico com os tempos médios de execução dos algoritmos na Figura 13.

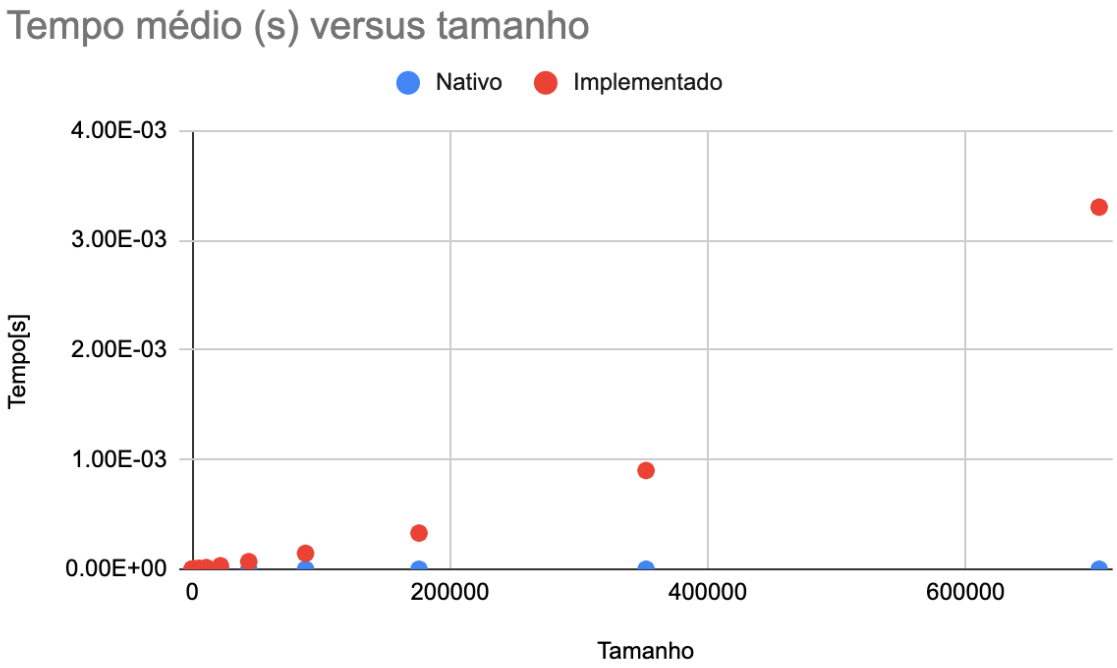

Figura 13: Gráfico comparando o tempo médio de execução das implementações da função String.*substring()* testadas sobre o *dataset* de frases aleatórias.

A partir da análise do gráfico, podemos concluir que o comportamento assintótico da curva de tempo médio da função nativa é diversas ordens de grandeza menor do que o da função implementada e, portanto, sua complexidade de pior caso é bem menor do que $O(n^2)$, e mais próxima de $O(n)$, sendo n a quantidade de caracteres da *String*. A complexidade alta da função implementada pode ser explicada pela imutabilidade do objeto *String* no JavaScript - a cada alteração feita nele, uma nova *String*, com o novo valor, é instanciada, e a antiga descartada. O método nativo garante que essa ineficiência do objeto não afete o tempo de execução.

6.3. String.toUpperCase()

O método *String.toUpperCase()* retorna uma nova String com todos os caracteres em letra maiúscula. Números e caracteres especiais não são afetados[17]. A sintaxe do método é apresentada no Algoritmo 33.

```
str.toUpperCase()
```

Algoritmo 33: sintaxe do método *String.substring()*.[17]

Apresentamos um exemplo de uso do método no Algoritmo 34.

```
> const a = 'minhastring123.';
> a.toUpperCase();
< 'MINHASTRING123.'
```

Algoritmo 34: exemplo de uso do método *String.substring()*

Como algoritmo de comparação, implementamos uma função que converte a *String* num *Array* de *chars* e percorre os elementos, um a um, verificando se ele é uma letra, e alterando para seu equivalente em caixa alta caso positivo. O algoritmo, de complexidade *O(n)*, é apresentado no Algoritmo 35.

```
function implementedToUpperCase(string) {
 arr = string.split('')                                  O(n)
 arr = arr.map(x => {                                    O(n)
   var charCode = x.charCodeAt(0);                         O(1)
   return charCode>=97 && charCode<=122 ?                  O(1)
String.fromCharCode(charCode - 32) : x;                    O(1)
 });
 return arr.join('');                                    Total: O(n)
}
```

Algoritmo 35: implementação de um algoritmo equivalente ao método *String.toUpperCase()* e sua respectiva análise de complexidade

Na Figura 14 apresentamos a curva do tempo médio de execução dos dois algoritmos.

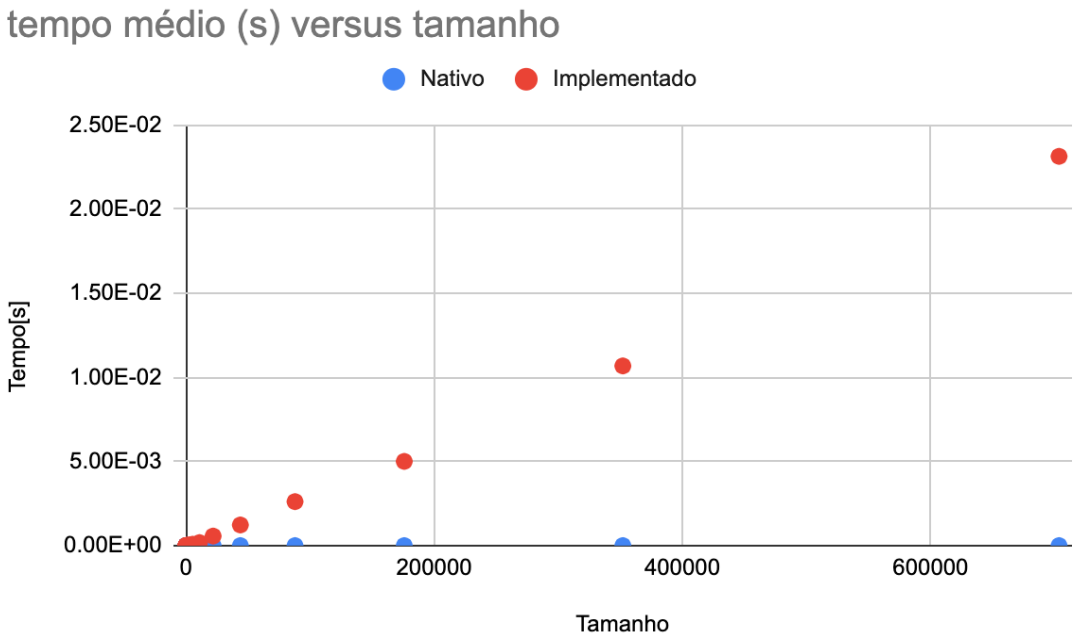

Figura 14: Gráfico comparando o tempo médio de execução das implementações da função String.*toUpperCase()* testadas sobre o *dataset* de frases aleatórias.

Nesta análise, observamos um resultado inesperado. O tempo de execução do método nativo de mostrou constante, independente do tamanho da *String* testada. Na Figura 15 mostramos apenas os pontos da curva de tempo médio de execução do método nativo.

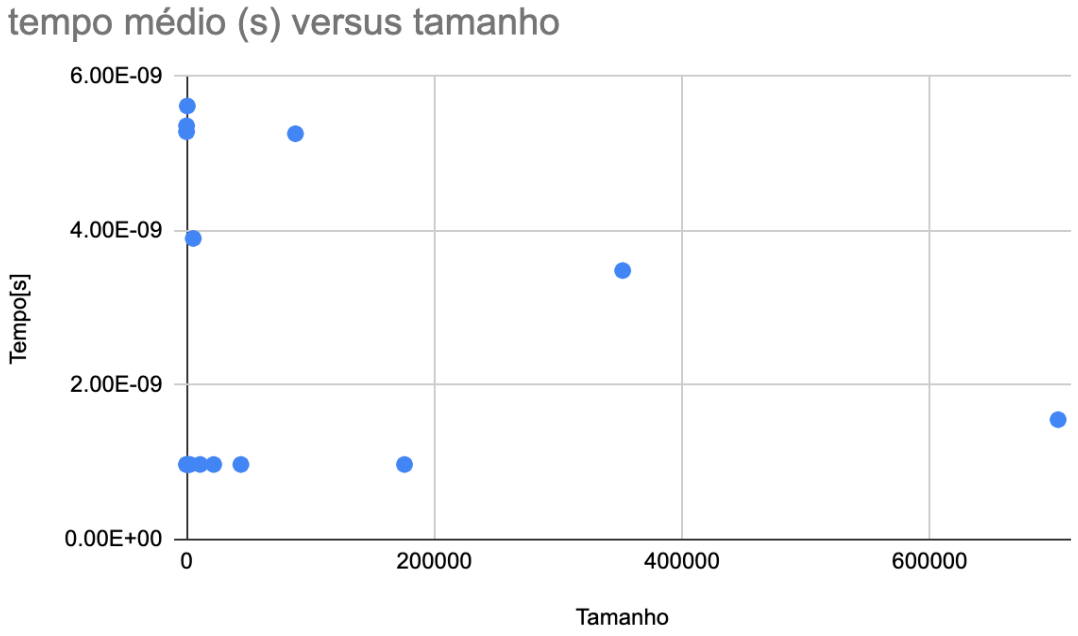

Figura 15: Curva do tempo médio de execução do método *String.toUpperCase()*

Como podemos ver, agora com mais detalhes, o tempo de execução médio do método é extremamente baixo (na casa dos 10^{-9} segundos) e é praticamente constante independente da quantidade de caracteres da *String*. Portanto, podemos afirmar que a complexidade de pior caso do método *String.toUpperCase()* é de *O(1)*.

7. Conclusão

A realização dos testes de desempenho dos métodos nativos do JavaScript implementados pelo Node.js e sua comparação com o desempenho de algoritmos similares de complexidade conhecida nos permitiu observar e afirmar, com base em evidências, alguns aspectos da complexidade desses métodos.

Primeiramente, podemos sumarizar as complexidades de tempo de pior caso encontradas para os métodos testados:

- *Array.concat():* ***O(n+m)***
- *Array.find():* ***O(n)***
- *Array.reverse():* ***O(n)***
- *Array.shift():* ***O(n)***
- *Array.unshift():* ***O(n)***
- *Array.sort():* ***O(n.log(n))***
- *Array.push():* ***O(n)***
- *String.match():* ***O(n)***
- *String.substring():* ***O(n)***
- *String.toUpperCase():* ***O(1)***

Tivemos dois resultados inesperados nos testes, nos métodos ***Array.sort()*** e ***String.toUpperCase()***.

No método ***Array.sort()*** pudemos observar como nem sempre a análise de complexidade de pior caso reflete bem a eficiência de um algoritmo - o *Quick Sort*, algoritmo de ordenação de complexidade de pior caso $O(n^2)$ testado, registrou o melhor

desempenho dentre todos os algoritmos testados, a despeito de sua complexidade maior. Este resultado levanta a importância também da análise de complexidade média de execução na hora de escolher o algoritmo mais eficiente a ser implementado.

Já no método ***String.toUpperCase()*** pudemos notar uma complexidade constante de *O(1)*, a despeito do algoritmo de melhor complexidade conhecido possuir complexidade de *O(n)*. Este método demonstra de forma mais exacerbada a vantagem de performance que podemos obter ao realizarmos otimizações a nível de compilador.

Por fim, descobrimos empiricamente que, embora geralmente otimizadas, a complexidade de tempo de pior caso dos algoritmos nativos não costumam ser diferentes do que a complexidade dos algoritmos mais comuns conhecidos que resolvem o mesmo problema. Isso nos permite afirmar que não há prejuízo na escolha das implementações nativas dos métodos testados neste trabalho versus suas implementações manuais.

8. Referências

[1] "Usage Statistics of JavaScript as Client-side Programming Language on Websites, July 2022", W3techs.com, 2022. [Online]. Disponível em: https://w3techs.com/technologies/details/cp-javascript/. [Acessado em: 21- Jul- 2022].

[2] "How Terra improved user engagement thanks to Dark Mode", web.dev, 2022. [Online]. Disponível em: https://web.dev/terra-dark-mode/. [Acessado em: 21- Jul- 2022].

[3] "How QuintoAndar increased conversion rates and pages per session by improving page performance", web.dev, 2022. [Online]. Disponível em: https://web.dev/quintoandar/. [Acessado em: 21- Jul- 2022].

[4] "Evaluating page experience for a better web | Google Search Central Blog | Google Developers", Google Developers, 2022. [Online]. Disponível em: https://developers.google.com/search/blog/2020/05/evaluating-page-experience. [Acessado em: 21- Jul- 2022].

[5] "Notação Big-O (artigo) | Algoritmos | Khan Academy", Khan Academy, 2022. [Online]. Disponível em: https://pt.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation. [Acessado em: 21- Jul- 2022].

[6] "Benchmark.js documentation", Benchmarkjs.com, 2022. [Online]. Disponível em: https://benchmarkjs.com/docs. [Acessado em: 21- Jul- 2022].

[7] "High Resolution Time", W3.org, 2022. [Online]. Disponível em: https://www.w3.org/TR/hr-time/#sec-DOMHighResTimeStamp. [Acessado em: 21- Jul- 2022].

[8] "Array.prototype.concat() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat. [Acessado em: 21- Jul- 2022].

[9] "Array.prototype.find() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find. [Acessado em: 21- Jul- 2022].

[10] "Array.prototype.reverse() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse. [Acessado em: 21- Jul- 2022].

[11] "Array.prototype.shift() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift. [Acessado em: 21- Jul- 2022].

[12] "Array.prototype.unshift() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift. [Acessado em: 21- Jul- 2022].

[13] "Array.prototype.sort() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort. [Acessado em: 21- Jul- 2022].

[14] "Array.prototype.push() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push. [Acessado em: 21- Jul- 2022].

[15] "String.prototype.match() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match. [Acessado em: 21- Jul- 2022].

[16] "String.prototype.substring() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substring. [Acessado em: 21- Jul- 2022].

[17] "String.prototype.toUpperCase() - JavaScript | MDN", Developer.mozilla.org, 2022. [Online]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase. [Acessado em: 21- Jul- 2022].