Aula 7 - Complexidade de Algoritmos: Introdução à Notação Big O

aTip Learn - Lógica de Programação com JavaScript

Objetivos

- Entender a importância da análise de complexidade de algoritmos
- Conhecer a notação Big O
- Compreender a diferença entre complexidade de tempo e espaço

Conteúdo

Introdução à Notação Big O

A notação Big O é uma ferramenta que nos permite analisar a eficiência de algoritmos em termos de espaço (memória) e tempo (quantidade de operações). Por meio da notação Big O podemos comparar diferentes algorithmos e escolher o mais adequado para um determinado problema.

Utilizando a notação Big O podemos descrever a eficiência de um algoritmo em relação ao tamanho da entrada desse algoritmo. Pense no tamanho da entrada como o número de elementos em um array passado como parâmetro para uma função, por exemplo. Com a notação Big O podemos descrever como o número de operações que um algoritmo realiza (ou a memória utilizada) cresce conforme o tamanho da entrada.

Veja abaixo dois exemplos de algoritmos, o primeiro recebe um vetor de números e retorna o valor do primeiro elemento do vetor. O segundo algoritmo recebe um vetor de números e retorna a soma de todos os elementos do vetor.

```
function primeiroElemento(vetor) {
  return vetor[0];
}

function somaElementos(vetor) {
  let soma = 0;
  for (let valor of vetor) {
    soma += valor;
  }
  return soma;
}

const vetor = [1, 2, 3, 4, 5];
console.log(primeiroElemento(vetor)); // 1
console.log(somaElementos(vetor)); // 15
```

Você saberia dizer qual algoritmo é mais eficiente em termos do número de operações (eficiência de tempo) realizadas?

No caso da chamada para a função **primeiroElemento** o número de operações é constante, não importa o tamanho do vetor, a função sempre realizará uma única operação para obter o primeiro elemento do vetor. Dizemos que a complexidade de tempo da função **primeiroElemento** é O(1). O número 1 na complexidade serve para dizer que o número de operações é constante, independente do tamanho da entrada.

Já a função **somaElementos** realiza um número de operações proporcional ao tamanho do vetor, ou seja, se o vetor tiver 10 elementos a função realizará 10 operações, se o vetor tiver 100 elementos a função realizará 100 operações. Dizemos que a complexidade de tempo da função **somaElementos** é O(n), onde n é o tamanho do vetor. Nesse caso dizemos que a complexidade do algoritmo é linear, pois o número de operações cresce linearmente com o tamanho da entrada.

Análise do Pior Caso

No exemplo anterior, vimos que a função **somaElementos** tem uma complexidade maior do que a função **primeiroElemento**. Agora imagine que o array de entrada passado para ambos os algoritmos possui apenas 1 elemento. Nesse caso, ambos os algoritmos realizarão uma única operação, poderíamos então dizer que a complexidade de ambos é a mesma?

A resposta é não, pois a notação Big O expressa a complexidade dos algoritmos no pior caso possível, que no caso dos algoritmos que vimos ocorreria com um array extremamente grande, ou com um número de elementos teoricamente infinito. Nesse caso a função **primeiroElemento** continuaria realizando uma única operação, enquanto a função **somaElementos** continuaria realizando um número de operações proporcional ao tamanho do vetor.

Comparação entre as Complexidades Comuns

O gráfico abaixo ilustra as principais complexidades que podemos encontrar em algoritmos:

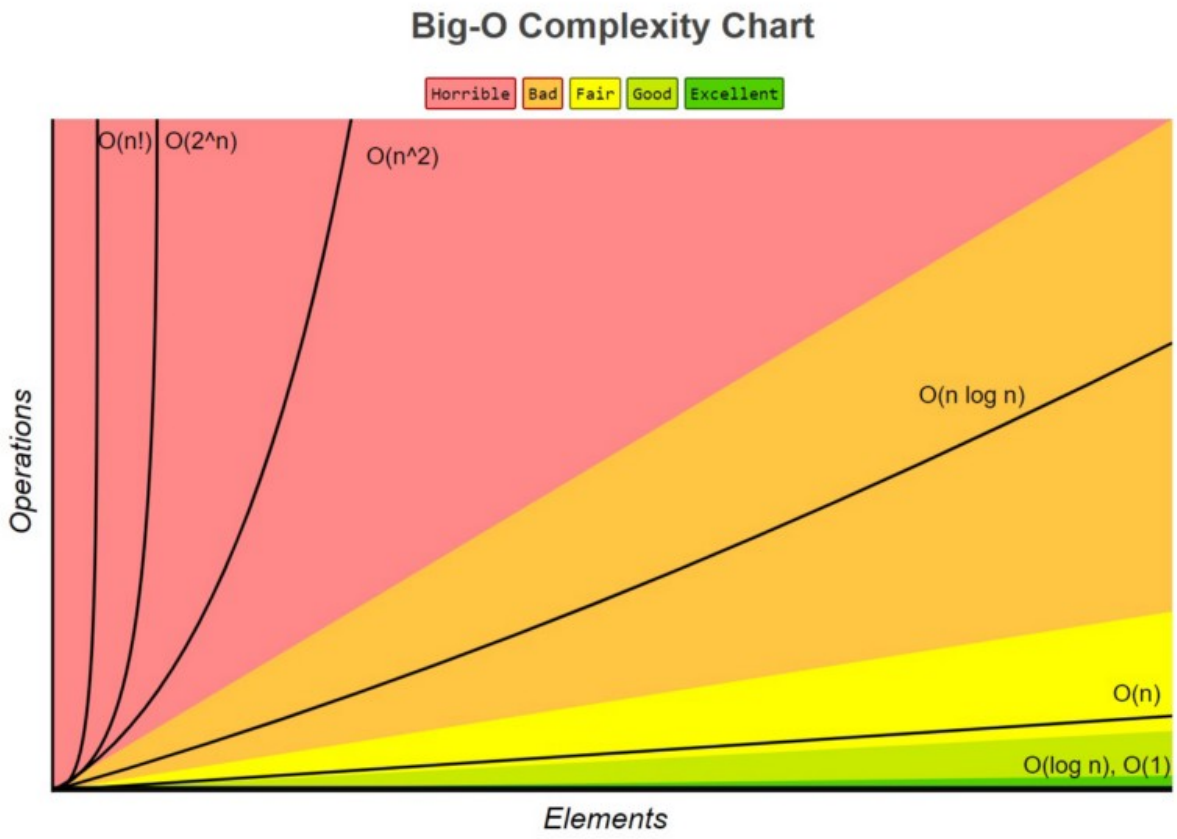

Figure 1: Tabela Big O

https://www.freecodecamp.org/portuguese/news/o-que-e-a-notacao-big-o-complexidade-de-tempo-e-de-espaco/

- O(1): Complexidade constante, o número de operações é constante, independente do tamanho da entrada. Esses algoritmos não costumam possuir laços de repetição. Exemplo: Acesso direto a um elemento de um array
- O(log n): Complexidade logarítmica, o número de operações cresce de forma logarítmica conforme o tamanho da entrada aumenta. Nesse tipo de algoritmo, a entrada é dividida pela metade a cada

iteração. Exemplo: Busca binária em um array ordenado.

- O(n): Complexidade linear, o número de operações cresce de forma proporcional ao tamanho da entrada. Esses algoritmos costumam possuir um laço de repetição que percorre todos os elementos da entrada. Exemplo: Percorrer um array para somar todos os elementos.
- O(n log n): Complexidade log-linear, o número de operações cresce de forma proporcional ao tamanho da entrada multiplicado pelo logaritmo do tamanho da entrada. Exemplo: Algoritmos de ordenação como merge sort e quick sort, onde para cada elemento da entrada é realizada uma operação logarítmica.
- O(n^2): Complexidade quadrática, o número de operações cresce de forma quadrática conforme o tamanho da entrada aumenta. Exemplo: Algoritmos de ordenação como bubble sort e insertion sort, onde para cada elemento da entrada é realizada uma operação linear.
- O(2^n): Complexidade exponencial, o número de operações cresce de forma exponencial conforme o tamanho da entrada aumenta. Exemplo: Algoritmos recursivos que geram uma árvore de recursão com 2 filhos para cada nó.
- O(n!): Complexidade fatorial, o número de operações cresce de forma fatorial conforme o tamanho da entrada aumenta. Exemplo: Algoritmos que geram todas as permutações de uma sequência de elementos.

Complexidade de Espaço

Até agora falamos apenas sobre a complexidade de tempo dos algoritmos, mas também podemos analisar a complexidade de espaço, ou seja, a quantidade de memória utilizada por um algoritmo. A complexidade de espaço é expressa da mesma forma que a complexidade de tempo, porém ao invés de analisar o número de operações realizadas, analisamos a quantidade de memória utilizada.

Um algoritmo com complexidade de espaço O(1) utiliza uma quantidade constante de memória, independente do tamanho da entrada, ou seja, o algoritmo não aloca novos espaços de memória conforme o tamanho da entrada aumenta. Já um algoritmo com complexidade de espaço O(n) aloca uma quantidade de memória proporcional ao tamanho da entrada, ou seja, se o tamanho da entrada é 10, o algoritmo aloca um espaço proporcional a 10, se o tamanho da entrada é 100, o algoritmo aloca um espaço proporcional a 100.

Exemplos de Algoritmos

Busca Linear

Dado um array de números, a função abaixo realiza uma busca linear para encontrar um determinado valor no array. A função retorna o índice do valor no array, ou -1 caso o valor não seja encontrado.

```
function buscaLinear(array, valor) {
  for (let i = 0; i < array.length; i++) {
    if (array[i] === valor) {
      return i;
    }
  }
  return -1;
}

const array = [2, 1, 3, 5, 4];
console.log(buscaLinear(array, 3)); // 2
console.log(buscaLinear(array, 6)); // -1
```

A complexidade de tempo da função **buscaLinear** é O(n), pois o número de operações cresce de forma proporcional ao tamanho do array. A complexidade de espaço da função **buscaLinear** é O(1), pois a função não aloca novos espaços de memória conforme o tamanho do array aumenta.

Busca Binária

Como podemos melhorar a performance do algoritmo de busca? Uma alternativa é utilizar a busca binária, que é um algoritmo mais eficiente para encontrar um valor em um array ordenado. A busca binária divide o espaço de busca pela metade a cada iteração, reduzindo o número de operações necessárias para encontrar um valor.

```
function buscaBinaria(array, valor) {
  let inicio = 0;
  let fim = array.length - 1;
  while (inicio <= fim) {
    let meio = Math.floor((inicio + fim) / 2);
    if (array[meio] === valor) {
      return meio;
    } else if (array[meio] < valor) {
      inicio = meio + 1;
    } else {
      fim = meio - 1;
    }
  }
  return -1;
}

const array = [1, 2, 3, 4, 5];
console.log(buscaBinaria(array, 3)); // 2
console.log(buscaBinaria(array, 6)); // -1
```

A complexidade de tempo da função **buscaBinaria** é O(log n), pois o número de operações cresce de forma logarítmica conforme o tamanho do array. A complexidade de espaço da função **buscaBinaria** é O(1), pois a função não aloca novos espaços de memória conforme o tamanho do array aumenta.

Por que O(log n)?

Dado um array ordenado, a busca binária aproveita-se deste fato para dividir o espaço de busca pela metade a cada iteração. Na primeira iteração o algoritmo verifica o elemento do meio do array, se o valor buscado for menor que o elemento do meio, o algoritmo descarta a metade superior do array, se o valor buscado for maior que o elemento do meio, o algoritmo descarta a metade inferior do array. Dessa forma, a cada iteração o algoritmo reduz pela metade o espaço de busca, até encontrar o valor ou determinar que o valor não está presente no array. Assim, em um array de 4 elementos, o algoritmo realiza no máximo 2 operações ($\log_2 4 = 2$), em um array de 8 elementos, o algoritmo realiza no máximo 3 operações ($\log_2 8 = 3$), e assim por diante.

Bubble Sort

O Bubble Sort é um algoritmo de ordenação simples que percorre o array diversas vezes, comparando elementos adjacentes e trocando de posição caso estejam fora de ordem. O algoritmo repete esse processo até que o array esteja ordenado.

```
function bubbleSort(array) {
  let n = array.length;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n - i - 1; j++) {
      if (array[j] > array[j + 1]) {
        let temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
```

```
    }
  }
  return array;
}

const array = [2, 1, 3, 5, 4];
console.log(bubbleSort(array)); // [1, 2, 3, 4, 5]
```

A complexidade de tempo do Bubble Sort é O(n^2), pois o número de operações cresce de forma quadrática conforme o tamanho do array. A complexidade de espaço do Bubble Sort é O(1), pois o algoritmo não aloca novos espaços de memória conforme o tamanho do array aumenta.

Por que O(n^2)?

Veja que a função bubbleSort possui dois laços de repetição aninhados, o laço externo percorre o array n vezes, o laço interno percorre o array n - i - 1 vezes. Apesar de o laço interno percorrer um número decrescente de elementos a cada iteração, na notação Big O consideramos apenas o termo de maior ordem, que no caso é n^2.

Como podemos melhorar a performance do Bubble Sort?

Um dos problemas com o Bubble Sort é que ele realiza diversas iterações mesmo que o array já esteja ordenado. Uma forma de melhorar a performance do Bubble Sort é verificarmos se houveram trocas durante uma iteração do laço externo, se não houveram trocas, significa que o array já está ordenado e podemos interromper o algoritmo.

```
function bubbleSortMelhorado(array) {
  let n = array.length;
  for (let i = 0; i < n; i++) {
    let trocou = false;
    for (let j = 0; j < n - i - 1; j++) {
      if (array[j] > array[j + 1]) {
        let temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
        trocou = true;
      }
    }
    if (!trocou) {
      return array;
    }
  }
  return array;
}
```

É importante ressaltar que mesmo com essa otimização, a complexidade de tempo do Bubble Sort continua sendo O(n^2), pois no pior caso o algoritmo ainda realizará n^2 operações. No entanto, a otimização pode melhorar a performance do algoritmo em casos onde o array já está parcialmente ordenado.