

PUC Minas  
Ciência da Computação

Mateus Henrique Medeiros Diniz

**Análise do desempenho do algoritmo QuickSort com diferentes estratégias de  
escolha de Pivô**

Belo Horizonte

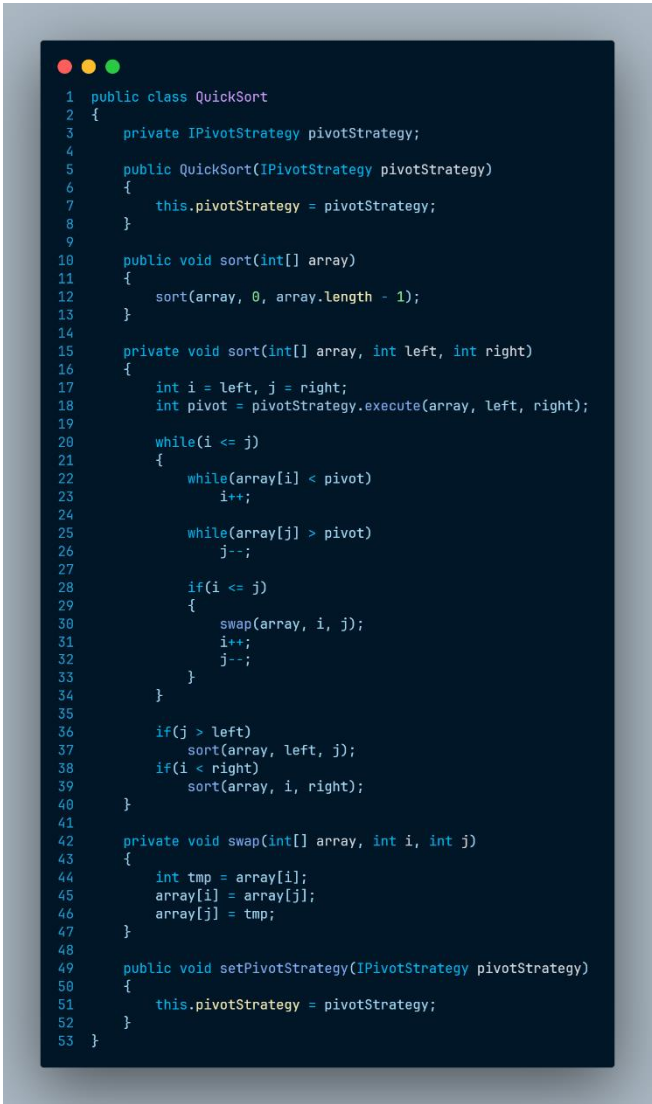
2024

## INTRODUÇÃO

Este trabalho tem como objetivo entender como diferentes estratégias de escolha do pivô do algoritmo de ordenação QuickSort afetam o seu desempenho.

## IMPLEMENTAÇÃO

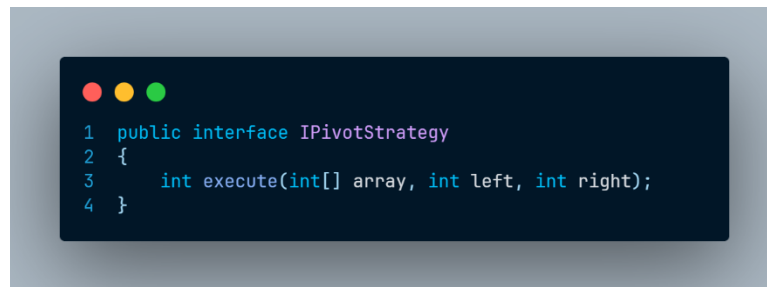
A implementação do algoritmo (Figura 1) e testes foram feitos utilizando a linguagem Java.



```
1 public class QuickSort
2 {
3     private IPivotStrategy pivotStrategy;
4
5     public QuickSort(IPivotStrategy pivotStrategy)
6     {
7         this.pivotStrategy = pivotStrategy;
8     }
9
10    public void sort(int[] array)
11    {
12        sort(array, 0, array.length - 1);
13    }
14
15    private void sort(int[] array, int left, int right)
16    {
17        int i = left, j = right;
18        int pivot = pivotStrategy.execute(array, left, right);
19
20        while(i <= j)
21        {
22            while(array[i] < pivot)
23                i++;
24
25            while(array[j] > pivot)
26                j--;
27
28            if(i <= j)
29            {
30                swap(array, i, j);
31                i++;
32                j--;
33            }
34        }
35
36        if(j > left)
37            sort(array, left, j);
38        if(i < right)
39            sort(array, i, right);
40    }
41
42    private void swap(int[] array, int i, int j)
43    {
44        int tmp = array[i];
45        array[i] = array[j];
46        array[j] = tmp;
47    }
48
49    public void setPivotStrategy(IPivotStrategy pivotStrategy)
50    {
51        this.pivotStrategy = pivotStrategy;
52    }
53 }
```

*Figura 1*

A classe QuickSort possui uma dependência de IPivotStrategy (Figura 2), uma abstração de uma estratégia de pivot, e a utiliza para encontrar o pivô quando necessário. A estratégia pode ser alternada a qualquer momento durante o tempo de execução

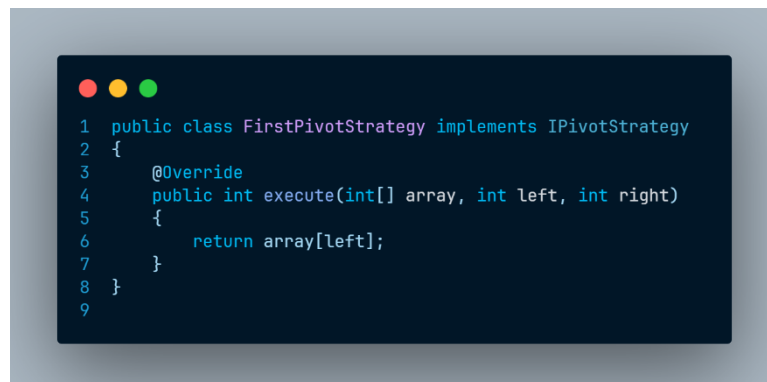


```
1 public interface IPivotStrategy
2 {
3     int execute(int[] array, int left, int right);
4 }
```

*Figura 2*

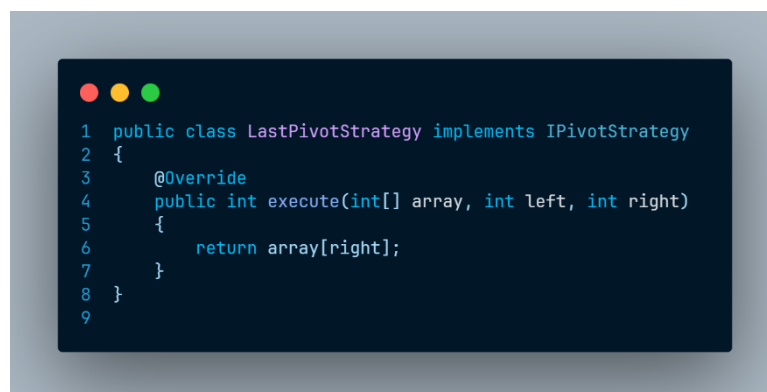
Para cada estratégia foi criada uma classe que implementa a interface IPivotStrategy.

Para as estratégias de escolher o primeiro item (Figura 3) e o último item (Figura 4) como pivô, basta retornar o primeiro ou último elemento com base nas variáveis left e right, que seguem os limites de atuação da instância do momento do método recursivo sort.



```
1 public class FirstPivotStrategy implements IPivotStrategy
2 {
3     @Override
4     public int execute(int[] array, int left, int right)
5     {
6         return array[left];
7     }
8 }
9
```

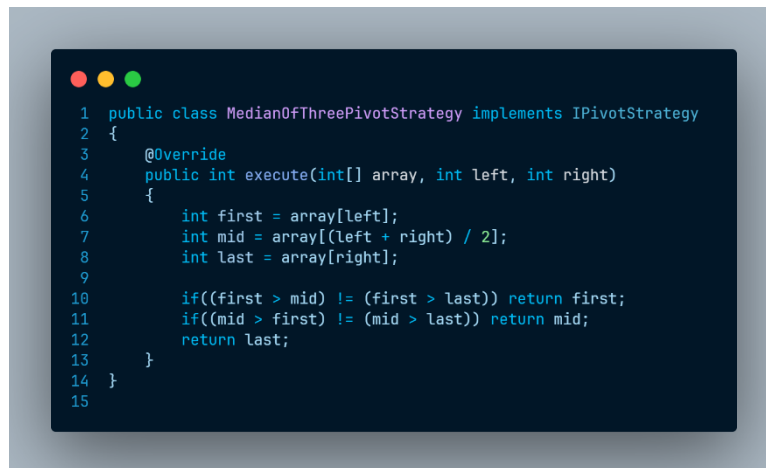
*Figura 3*



```
1 public class LastPivotStrategy implements IPivotStrategy
2 {
3     @Override
4     public int execute(int[] array, int left, int right)
5     {
6         return array[right];
7     }
8 }
9
```

*Figura 4*

Para a estratégia de escolher o pivô a partir da mediana do primeiro, central e último item do array (Figura 5), também é feito uso dos delimitadores left e right, além de algumas condicionais para encontrar o item central.

A screenshot of a code editor showing the implementation of the MedianOfThreePivotStrategy class. The code is in Java and implements the IPivotStrategy interface. It defines an execute method that takes an array and left and right indices. Inside the method, it calculates the first, mid, and last elements. Then, it uses conditional logic to determine the median of these three elements and returns it as the pivot. The code is numbered from 1 to 15.

```
1 public class MedianOfThreePivotStrategy implements IPivotStrategy
2 {
3     @Override
4     public int execute(int[] array, int left, int right)
5     {
6         int first = array[left];
7         int mid = array[(left + right) / 2];
8         int last = array[right];
9
10        if((first > mid) != (first > last)) return first;
11        if((mid > first) != (mid > last)) return mid;
12        return last;
13    }
14 }
15
```

*Figura 5*

Para a estratégia de escolher o pivô aleatoriamente (Figura 6), é escolhido um número utilizando a classe nativa Random entre left e right (inclusivo) que corresponde ao índice do item escolhido. Cada chamada do método muda a semente para obter resultados diferentes.


A screenshot of a code editor showing the implementation of the RandomPivotStrategy class. The code is in Java and implements the IPivotStrategy interface. It defines a constructor that initializes a Random object. The execute method uses the Random object to select a random index between left and right (inclusive) and returns the element at that index. The code is numbered from 1 to 19.

```
1 import java.util.Random;
2
3 public class RandomPivotStrategy implements IPivotStrategy
4 {
5     private Random random;
6
7     public RandomPivotStrategy()
8     {
9         random = new Random();
10    }
11
12    @Override
13    public int execute(int[] array, int left, int right)
14    {
15        random.setSeed(System.currentTimeMillis());
16
17        return array[left + random.nextInt(right - left + 1)];
18    }
19 }
```

*Figura 6*

Partindo para a implementação dos testes, foram gerados arrays ordenados, quase ordenados e aleatórios, todos com tamanhos variados, para serem ordenados.

Em todos os arrays ordenados (Figura 7), os valores de cada índice correspondem à sua posição no array, a partir do 0.



```
1 static int[] generateOrderedArray(int size)
2 {
3     int[] array = new int[size];
4
5     for(int i = 0; i < size; i++)
6     {
7         array[i] = i;
8     }
9
10    return array;
11 }
```

*Figura 7*

Para os arrays quase ordenados (Figura 8), foi reaproveitada a função de geração de arrays ordenados. Na implementação, são trocadas as posições de 2 elementos entre si de forma aleatória  $\frac{N}{10}$  (onde  $N$  é o tamanho do array) vezes, de forma a aleatorizar 10% do array.



```
1 static int[] generateAlmostOrderedArray(int size)
2 {
3     int[] array = generateOrderedArray(size);
4
5     Random random = new Random(System.currentTimeMillis());
6
7     int swapCount = size / 10;
8
9     for(int i = 0; i < swapCount; i++)
10    {
11        int index1 = random.nextInt(size);
12        int index2 = random.nextInt(size);
13
14        int tmp = array[index1];
15        array[index1] = array[index2];
16        array[index2] = tmp;
17    }
18
19    return array;
20 }
```

*Figura 8*

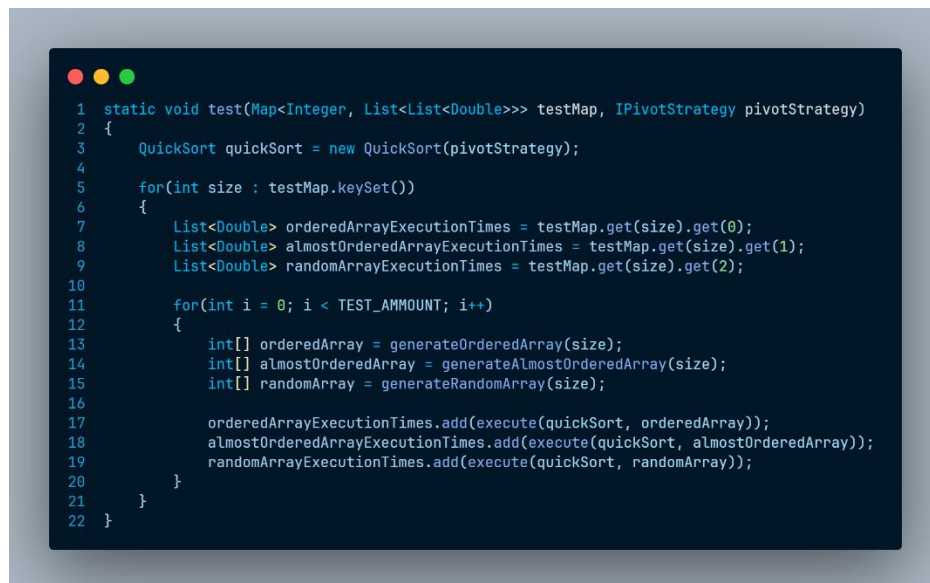
Para os arrays aleatórios (Figura 9), um número aleatório entre no intervalo  $[0, N)$  é inserido em cada posição do array.

A screenshot of a code editor with a dark background and light-colored text. The code is in Java and defines a static method named generateRandomArray. The method takes an integer parameter 'size' and returns an integer array. It uses a Random object to generate random integers for each element in the array.

```
1 static int[] generateRandomArray(int size)
2 {
3     Random random = new Random(System.currentTimeMillis());
4
5     int[] array = new int[size];
6
7     for(int i = 0; i < size; i++)
8     {
9         array[i] = random.nextInt(size);
10    }
11
12    return array;
13 }
```

*Figura 9*

Foram registrados os tempos de execução do algoritmo para cada estratégia de escolha do pivô, para cada tipo de array nos tamanhos 100, 1000 e 10000 (Figura 10). Esse processo se repete 10 vezes, e apenas a média final (Figura 11) entre os todos os tempos para cada variação da execução será considerada.

A screenshot of a code editor with a dark background and light-colored text. The code is in Java and defines a static method named test. The method takes a Map<Integer, List<List<Double>>> testMap, an IPivotStrategy pivotStrategy, and returns void. It creates a QuickSort object and iterates over the sizes in testMap.keySet(). For each size, it generates three arrays: ordered, almostOrdered, and random. It then measures the execution time of the QuickSort algorithm for each array type and adds the results to the corresponding list in testMap.

```
1 static void test(Map<Integer, List<List<Double>>> testMap, IPivotStrategy pivotStrategy)
2 {
3     QuickSort quickSort = new QuickSort(pivotStrategy);
4
5     for(int size : testMap.keySet())
6     {
7         List<Double> orderedArrayExecutionTimes = testMap.get(size).get(0);
8         List<Double> almostOrderedArrayExecutionTimes = testMap.get(size).get(1);
9         List<Double> randomArrayExecutionTimes = testMap.get(size).get(2);
10
11         for(int i = 0; i < TEST_AMMOUNT; i++)
12         {
13             int[] orderedArray = generateOrderedArray(size);
14             int[] almostOrderedArray = generateAlmostOrderedArray(size);
15             int[] randomArray = generateRandomArray(size);
16
17             orderedArrayExecutionTimes.add(execute(quickSort, orderedArray));
18             almostOrderedArrayExecutionTimes.add(execute(quickSort, almostOrderedArray));
19             randomArrayExecutionTimes.add(execute(quickSort, randomArray));
20         }
21     }
22 }
```

*Figura 10*

```

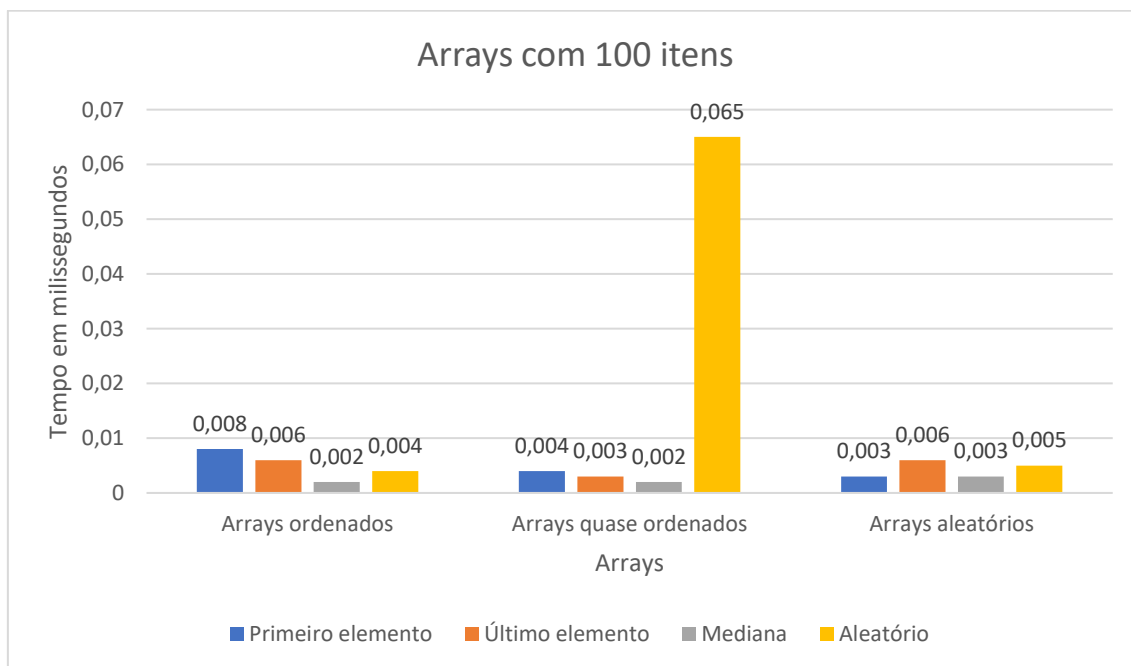
1 static double calculateAverage(List<Double> times)
2 {
3     double sum = 0;
4     for (double time : times)
5     {
6         sum += time;
7     }
8     return sum / times.size();
9 }

```

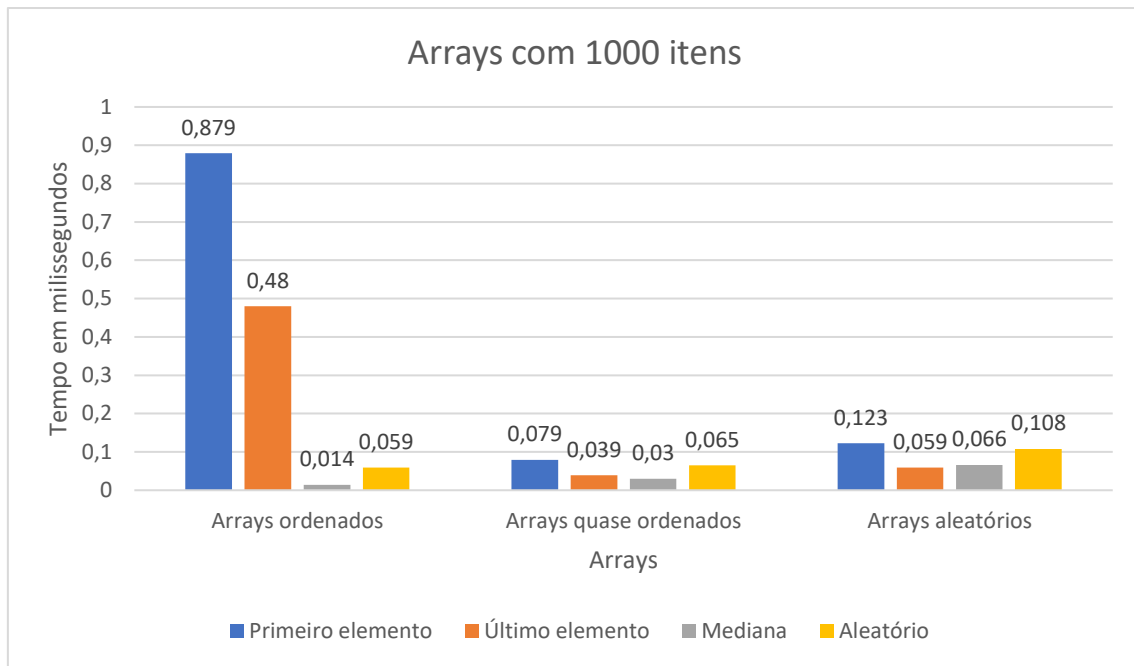
*Figura 11*

## ANÁLISE DO DESEMPENHO

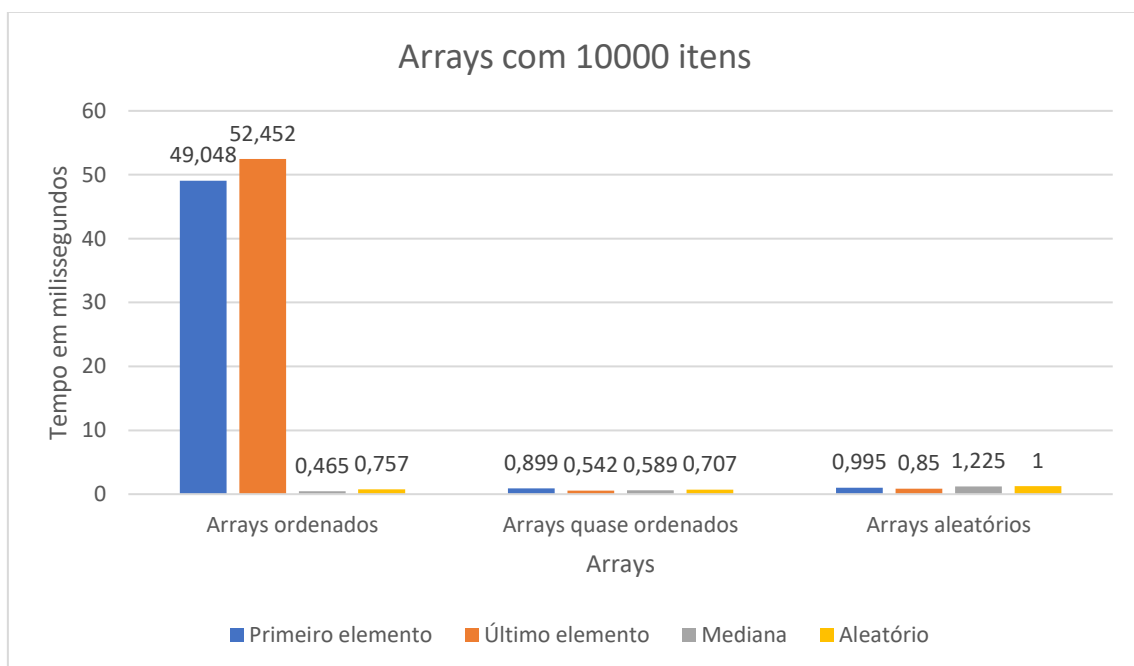
Após a efetuação dos testes, foram obtidos os seguintes resultados, organizados em gráficos por tamanho do array.



*Figura 12*



*Figura 13*



*Figura 14*

Há uma diferença gritante entre escolher o primeiro ou último elemento do vetor como pivô e as outras duas estratégias, principalmente nos arrays de tamanho 1000 e 10000, quando o array está ordenado. Pode-se assumir, ainda, que para tamanhos maiores, a diferença ficará mais acentuada. Essas estratégias não são recomendadas quando o array está ordenado.

Para os arrays quase ordenados, a diferença entre as quatro estratégias foi menor. Destaca-se o pior desempenho da escolha do pivô aleatório para o array de 100 itens.



Isso porém, deve-se ao fato da escolha do pivô ser aleatorizada. As demais estratégias dependem de escolher valores favoráveis para pivôs em um array 90% ordenado. Mesmo assim, a mediana de 3 elementos resultou em execuções mais rápidas nos dois primeiros casos, superada pela escolha do último pivô apenas no 3º, e em todos os casos o primeiro elemento como pivô resultou em execuções lentas.

Para os arrays aleatórios, a diferença entre as estratégias também foi mínima. O destaque da vez vai para a mediana de 3 elementos, que resultou nas execuções mais lentas para arrays maiores.

## CONCLUSÃO

A partir das análises, é possível perceber que a escolha do primeiro ou do último elemento do array como pivô em arrays ordenados resulta no pior caso do QuickSort:  $\theta(N^2)$ . Quando a escolha do pivô é aleatorizada, o tempo de execução se encontra, na maioria das vezes, entre o pior e melhor caso. Isso se deve ao fato de que a probabilidade da escolha do pivô corresponder aos extremos é mínima.

A mediana entre o primeiro, central e último elemento do array, em contrapartida, mostrou um comportamento curioso. Para arrays ordenados e quase ordenados, a estratégia se mostrou a mais eficiente na grande maioria dos casos. Para arrays aleatórios, porém, houve um crescimento acelerado do tempo de execução conforme o tamanho do array aumentava. Nesse caso, todas as estratégias acabam por escolher valores aleatórios. Uma possível suposição é que candidatar 3 valores aleatórios para serem o pivô é geralmente pior do que escolher um diretamente.

Então, pode-se concluir que, para arrays ordenados e quase ordenados, é mais provável obter uma execução eficiente com a estratégia da mediana de 3 elementos. Para arrays aleatórios, porém, pode ser melhor optar por estratégias que escolham diretamente um dos elementos do array. Apesar da conclusão estar alinhada com os resultados obtidos, em uma mediana de 3 elementos, de forma lógica, a probabilidade de escolher o elemento que levará ao melhor caso é maior.