Laboratório 06 - Quicksort e seu pivô

Aluno: Artur Bomtempo Colen

Código implementado

```java
import java.util.Random;


/**
* LAB06 - Quicksort e seu pivô
*
* @author Artur Bomtempo Colen
* @version 1.0, 07/10/2024
*/


public class QuickSortVariations {
   /**
    * Troca os elementos nas posições i e j em um array.
    *
    * @param i o índice do primeiro elemento a ser trocado
    * @param j o índice do segundo elemento a ser trocado
    * @param array o array no qual os elementos serão trocados
    */
   public static void swap(int i, int j, int[] array) {
       int temp = array[i];
       array[i] = array[j];
       array[j] = temp;
   }


   /**
     * Ordena o array usando o algoritmo QuickSort, escolhendo o primeiro elemento como pivô.
     * O processo é recursivo e continua até que a sublista tenha menos de dois elementos.
    *
    * @param array o array a ser ordenado
    * @param left  o índice inicial da sublista
    * @param right o índice final da sublista
    */
```

```java
    public static void QuickSortFirstPivot(int[] array, int left, int right) {
        if (left < right) {
            int pivotIndex = partitionFirstPivot(array, left, right);
            QuickSortFirstPivot(array, left, pivotIndex - 1);
            QuickSortFirstPivot(array, pivotIndex + 1, right);
        }
    }

    /**
     * Particiona o array tomando o primeiro elemento como pivô.
     * Elementos menores que o pivô são movidos para a esquerda, e os maiores para a direita.
     * Ao final, o pivô é colocado em sua posição correta e o índice do pivô é retornado.
     *
     * @param array o array a ser particionado
     * @param left  o índice inicial da sublista
     * @param right o índice final da sublista
     * @return o índice final do pivô após a partição
     */
    private static int partitionFirstPivot(int[] array, int left, int right) {
        int pivot = array[left];
        int i = left + 1;

        for (int j = left + 1; j <= right; j++) {
            if (array[j] < pivot) {
                swap(i, j, array);
                i++;
            }
        }
        swap(left, i - 1, array);
        return i - 1;
    }

    /**
     * Ordena o array usando o algoritmo QuickSort, escolhendo o último elemento como pivô.
```

```
     * O processo é recursivo e continua até que a sublista tenha menos de dois elementos.
     *
     * @param array o array a ser ordenado
     * @param left  o índice inicial da sublista
     * @param right o índice final da sublista
     */
    public static void QuickSortLastPivot(int[] array, int left, int right) {
        if (left < right) {
            int pivotIndex = partitionLastPivot(array, left, right);
            QuickSortLastPivot(array, left, pivotIndex - 1);
            QuickSortLastPivot(array, pivotIndex + 1, right);
        }
    }

    /**
     * Particiona o array tomando o último elemento como pivô.
     * Elementos menores que o pivô são movidos para a esquerda, e os maiores para a direita.
     * Ao final, o pivô é colocado em sua posição correta e o índice do pivô é retornado.
     *
     * @param array o array a ser particionado
     * @param left  o índice inicial da sublista
     * @param right o índice final da sublista (que contém o pivô)
     * @return o índice final do pivô após a partição
     */
    private static int partitionLastPivot(int[] array, int left, int right) {
        int pivot = array[right];
        int i = left;

        for (int j = left; j < right; j++) {
            if (array[j] < pivot) {
                swap(i, j, array);
                i++;
            }
        }
```

```
        swap(i, right, array);
        return i;
    }

    /**
     * Ordena o array usando o algoritmo QuickSort, escolhendo um pivô aleatório.
      * O processo é recursivo e continua até que a sublista tenha menos de dois elementos.
     *
     * @param array o array a ser ordenado
     * @param left  o índice inicial da sublista
     * @param right o índice final da sublista
     */
    public static void QuickSortRandomPivot(int[] array, int left, int right) {
        if (left < right) {
            int pivotIndex = partitionRandomPivot(array, left, right);
            QuickSortRandomPivot(array, left, pivotIndex - 1);
            QuickSortRandomPivot(array, pivotIndex + 1, right);
        }
    }

    /**
     * Particiona o array escolhendo um pivô aleatório.
      * O pivô é trocado com o último elemento, e a partição é realizada em relação a ele.
     * O índice do pivô é retornado após a partição.
     *
     * @param array o array a ser particionado
     * @param left  o índice inicial da sublista
     * @param right o índice final da sublista (que contém o pivô)
     * @return o índice final do pivô após a partição
     */
    private static int partitionRandomPivot(int[] array, int left, int right) {
        Random rand = new Random();
        int pivotIndex = left + rand.nextInt(right - left + 1);
        swap(pivotIndex, right, array);
        return partitionLastPivot(array, left, right);
```

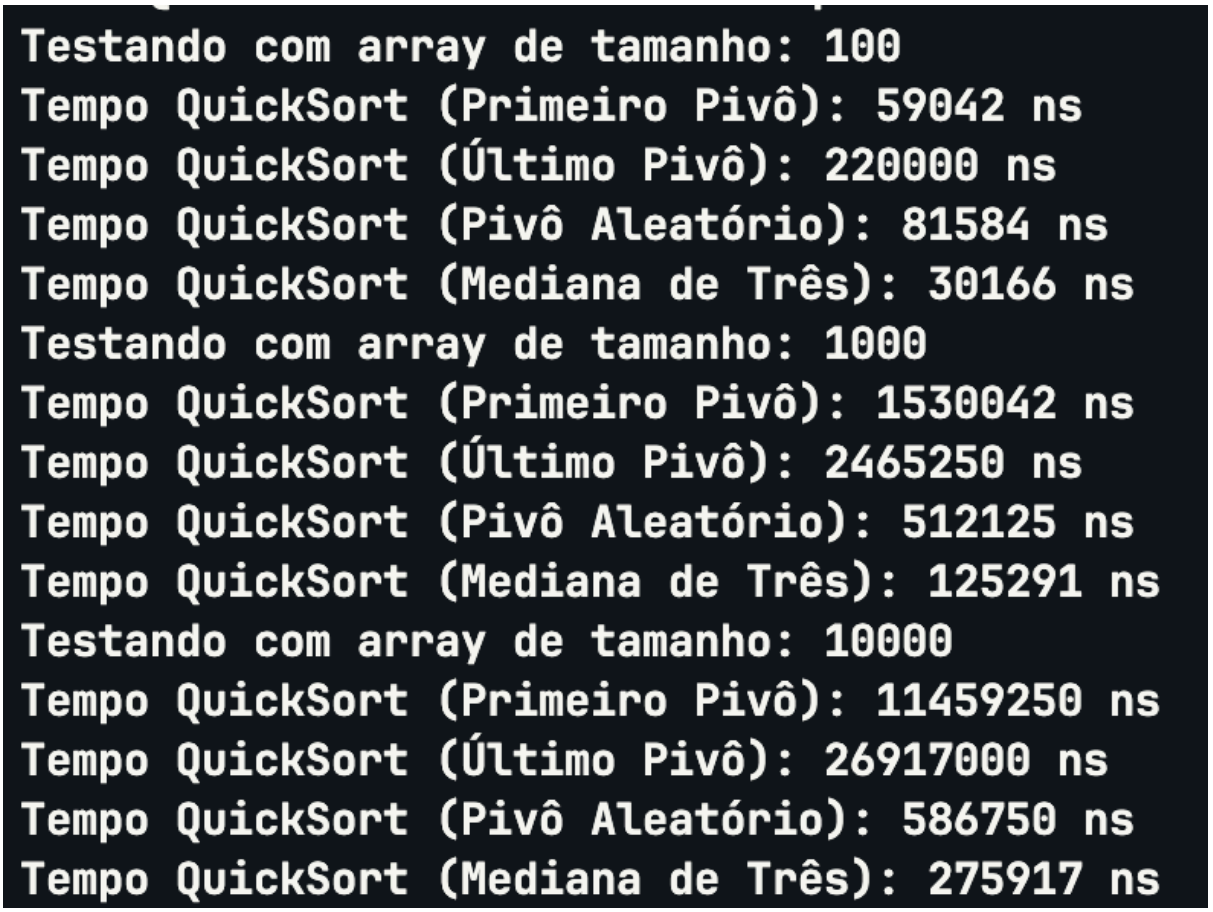
```java
    }

    /**
     * Ordena o array usando o algoritmo QuickSort, escolhendo a mediana de três elementos como pivô.
     * O processo é recursivo e continua até que a sublista tenha menos de dois elementos.
     *
     * @param array o array a ser ordenado
     * @param left  o índice inicial da sublista
     * @param right o índice final da sublista
     */
    public static void QuickSortMedianOfThree(int[] array, int left, int right) {
        if (left < right) {
            int pivotIndex = partitionMedianOfThree(array, left, right);
            QuickSortMedianOfThree(array, left, pivotIndex - 1);
            QuickSortMedianOfThree(array, pivotIndex + 1, right);
        }
    }

    /**
     * Particiona o array escolhendo a mediana de três elementos (início, meio e fim) como pivô.
     * O pivô é trocado com o último elemento, e a partição é realizada em relação a ele.
     * O índice do pivô é retornado após a partição.
     *
     * @param array o array a ser particionado
     * @param left  o índice inicial da sublista
     * @param right o índice final da sublista (que contém o pivô)
     * @return o índice final do pivô após a partição
     */
    private static int partitionMedianOfThree(int[] array, int left, int right) {
        int mid = (left + right) / 2;
        int pivot = medianOfThree(array, left, mid, right);
        swap(pivot, right, array);
        return partitionLastPivot(array, left, right);
```

```
    }

    /**
     * Determina a mediana entre três elementos do array.
     * Os elementos são identificados pelos índices fornecidos (a, b e c).
     * O método reorganiza esses elementos no array para garantir que o do meio seja a mediana.
     *
     * @param array o array que contém os elementos
     * @param a o índice do primeiro elemento
     * @param b o índice do segundo elemento
     * @param c o índice do terceiro elemento
     * @return o índice do elemento que é a mediana entre os três
     */
    private static int medianOfThree(int[] array, int a, int b, int c) {
        if (array[a] > array[b]) {
            swap(a, b, array);
        }
        if (array[b] > array[c]) {
            swap(b, c, array);
        }
        if (array[a] > array[b]) {
            swap(a, b, array);
        }
        return b;
    }

    /**
     * Método principal que executa testes de desempenho das diferentes implementações do algoritmo QuickSort.
     * O programa cria arrays de tamanhos variados (100, 1000, 10000) e testa cada variação de escolha de pivô.
     * Os tempos de execução são medidos e exibidos para cada implementação.
     */
    public static void main(String[] args) {
        int[] sizes = {100, 1000, 10000};
        Random rand = new Random();
```

```
        for (int size : sizes) {
            int[] arrayOrdered = new int[size];
            int[] arrayRandom = new int[size];
            int[] arrayAlmostSorted = new int[size];

            for (int i = 0; i < size; i++) {
                arrayOrdered[i] = i;
                arrayRandom[i] = rand.nextInt(size);
                arrayAlmostSorted[i] = i;

                if (i % 10 == 0) {
                    arrayAlmostSorted[i] = rand.nextInt(size);
                }
            }

            System.out.println("Testando com array de tamanho: " + size);

            long startTime, endTime;

            int[] arrayTest = arrayOrdered.clone();

            startTime = System.nanoTime();
            QuickSortFirstPivot(arrayTest, 0, arrayTest.length - 1);
            endTime = System.nanoTime();
                System.out.println("Tempo QuickSort (Primeiro Pivô): " + (endTime -
startTime) + " ns");

            arrayTest = arrayOrdered.clone();
            startTime = System.nanoTime();
            QuickSortLastPivot(arrayTest, 0, arrayTest.length - 1);
            endTime = System.nanoTime();
                 System.out.println("Tempo QuickSort (Último Pivô): " + (endTime -
startTime) + " ns");

            arrayTest = arrayOrdered.clone();
            startTime = System.nanoTime();
```

```
            QuickSortRandomPivot(arrayTest, 0, arrayTest.length - 1);
            endTime = System.nanoTime();
               System.out.println("Tempo QuickSort (Pivô Aleatório): " + (endTime -
startTime) + " ns");

            arrayTest = arrayOrdered.clone();
            startTime = System.nanoTime();
            QuickSortMedianOfThree(arrayTest, 0, arrayTest.length - 1);
            endTime = System.nanoTime();
              System.out.println("Tempo QuickSort (Mediana de Três): " + (endTime -
startTime) + " ns");
        }
    }
}
```

Resultado obtido:



Relatório

Funcionamento das Estratégias de Escolha do Pivô

1. **Primeiro Pivô:**

A estratégia de primeiro pivô seleciona o primeiro elemento do array como pivô. Essa abordagem é simples, porém suscetível a escolher pivôs não representativos, o que pode levar a uma má distribuição dos elementos em subarrays, especialmente se o array já estiver ordenado ou quase ordenado, o que piora o desempenho.

2. **Último Pivô:**

Semelhante à escolha do primeiro pivô, essa estratégia seleciona o último elemento do array como pivô. Embora ligeiramente diferente, os problemas são os mesmos que o primeiro pivô, resultando em má eficiência quando o array está ordenado ou quase ordenado.

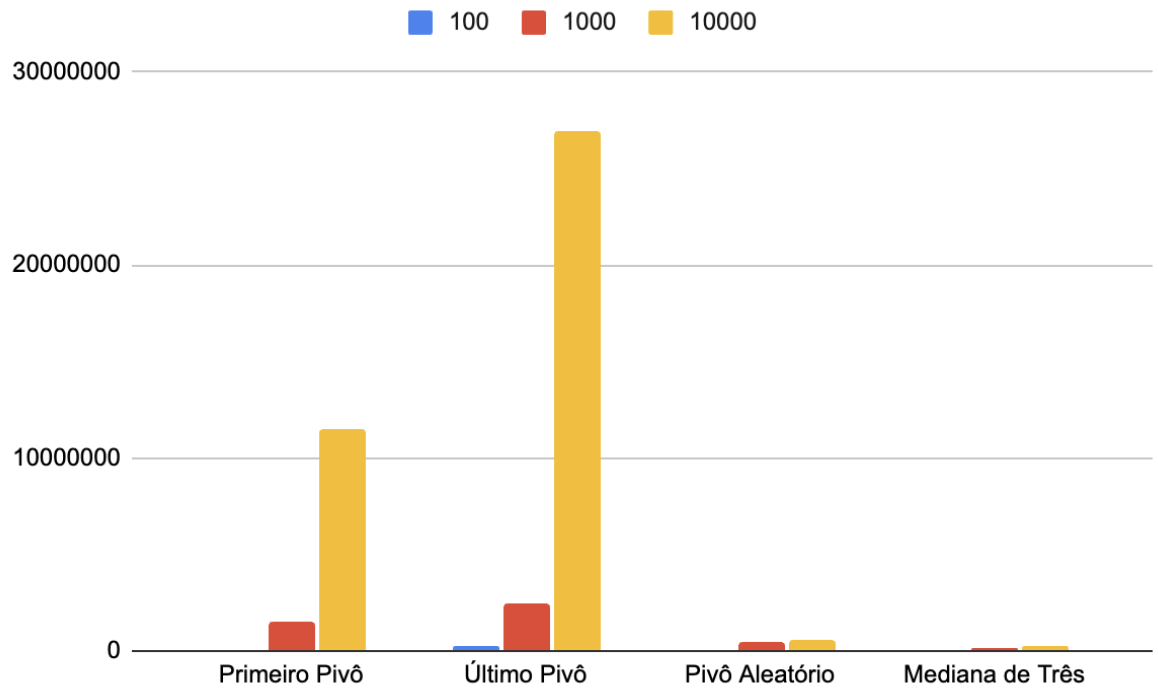
3. **Pivô Aleatório:**

Nessa estratégia, um pivô é escolhido aleatoriamente de qualquer posição dentro do subarray. A aleatoriedade ajuda a evitar os casos piores típicos dos dois métodos anteriores, pois aumenta a chance de o pivô ser representativo da distribuição dos dados.

4. **Mediana de Três:**

Essa técnica escolhe o pivô como a mediana entre o primeiro, o último e o elemento do meio do array. Essa estratégia visa reduzir a probabilidade de escolher um pivô ruim, oferecendo melhor estabilidade no tempo de execução, principalmente em arrays já ordenados ou quase ordenados.

Gráfico de Tempo de Execução



O gráfico gerado apresenta uma comparação do tempo de execução das variações do algoritmo QuickSort (Primeiro Pivô, Último Pivô, Pivô Aleatório, Mediana de Três) em três diferentes tamanhos de arrays: 100, 1000 e 10.000 elementos. As cores azul, vermelho e amarelo representam, respectivamente, o tempo de execução para arrays de 100, 1000 e 10.000 elementos.

Discussão sobre a Eficiência das Estratégias

- **Melhor Desempenho:** A Mediana de Três se mostrou a mais eficiente em todos os cenários testados, independentemente do tamanho do array. Isso ocorre porque essa estratégia minimiza a escolha de um pivô ruim, criando subarrays de tamanhos mais equilibrados e evitando, na maioria das vezes, os piores casos de complexidade.
- **Bom Desempenho com Aleatoriedade:** O Pivô Aleatório também apresentou tempos de execução bons, muito mais rápidos que as abordagens de primeiro e último pivô. A aleatoriedade reduz a probabilidade de um pivô mal escolhido, oferecendo boa eficiência para arrays desordenados.
- **Desempenho Ruim com Primeiro e Último Pivô:** Tanto o Primeiro Pivô quanto o Último Pivô tiveram um desempenho muito inferior. Especialmente para arrays ordenados ou quase ordenados, essas estratégias resultam em partições desbalanceadas, o que causa uma complexidade $O(n^2)$ no pior caso, em vez da esperada $O(n \log n)$.

Conclusão

A estratégia da **Mediana de Três** é a mais eficiente e estável em diferentes cenários, enquanto o Pivô Aleatório também se comporta bem. As abordagens de Primeiro Pivô e Último Pivô são menos eficientes e devem ser evitadas em contextos onde o array pode estar ordenado ou quase ordenado, pois tendem a causar os piores casos de desempenho do algoritmo QuickSort.