



Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática
Disciplina: Algoritmo e Estrutura de Dados II
Atividade: Laboratório 06 - Quicksort e seu pivô

Nome: Davi Cândido de Almeida _1527368

Relatório laboratório 06 - Quicksort e seu pivô

Link de acesso ao meu GitHub completo:

https://github.com/DaviKandido/Algoritmos_e_Estrutura_de_Dados_II

O arquivo da atividade se encontrar em /LABS_Exercicios/lab06:

https://github.com/DaviKandido/Algoritmos_e_Estrutura_de_Dados_II/tree/main/LABS_Exercicios/lab06

O arquivo principal: [VariacaoPivoQuickshort.java](#)

Tópico 0 - QuickSort e o funcionamento de cada estratégia de escolha de pivô

Antes é necessário demonstrar o funcionamento do método de QuickSort comum e as alterações implementadas em cada um dos diferentes métodos de escolha de pivô:

```
1  private void quicksort(int esq, int dir) {
2      int i = esq, j = dir;
3      int pivo = array[(dir+esq)/2];
4      while (i <= j) {
5          while (array[i] < pivo) i++;
6          while (array[j] > pivo) j--;
7          if (i <= j) {
8              swap(i, j);
9              i++;
10             j--;
11         }
12     }
13     if (esq < j) quicksort(esq, j);
14     if (i < dir) quicksort(i, dir);
15 }
```

Método de ordenação QuickSort comum - imagem 1

O método comum parte do princípio de escolha de um pivô baseado na posição intermediária a parcela a se ordenar, no entanto essa implementação pode acarretar em escolhas ineficientes de pivô o que consequentemente implica em um algoritmo de ordenação menos eficiente

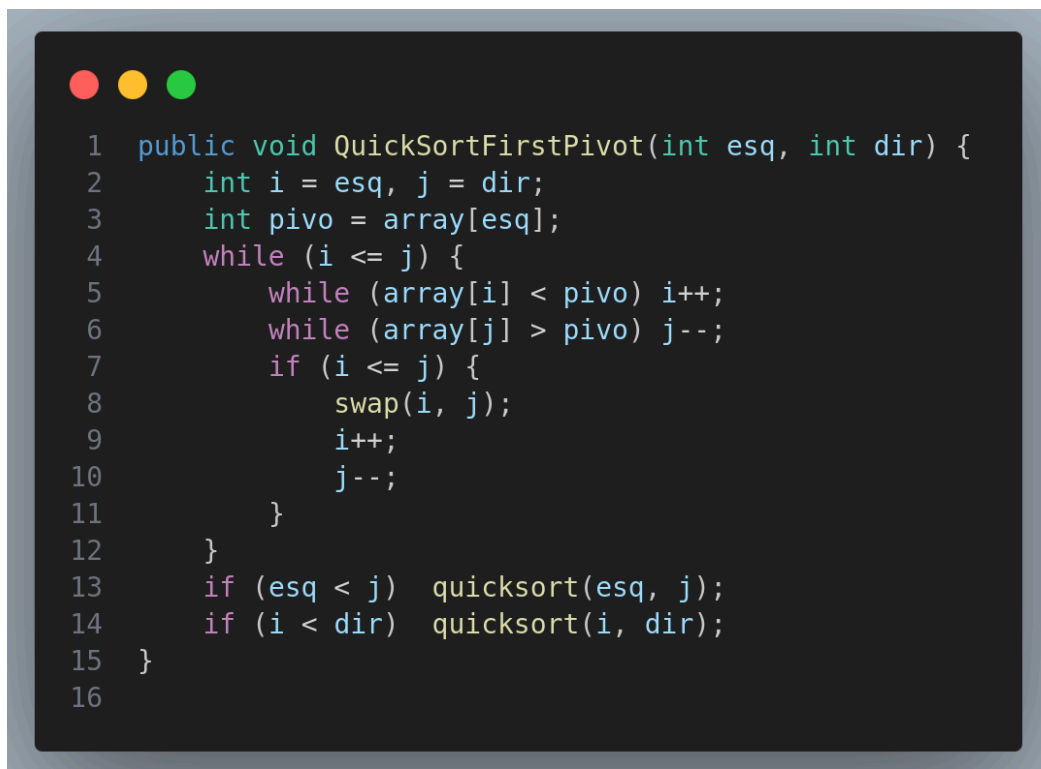
Alguns métodos implementados foram:

- Escolha da primeira posição do array como pivô - QuickSortLastPivot();
- Escolha da última posição do array como pivô - QuickSortLastPivot();
- Escolha de uma posição aleatória é válida dentro do array - QuickSortRandomPivot();
- E a mediana entre o primeiro elemento, o elemento central e o último elemento - QuickSortMedianOfThree ();

Vejamos cada um deles e algumas possíveis implicações ou benefícios:

Escolha da primeira posição do array como pivô - QuickSortLastPivot();

O **QuickSortLastPivot** se baseia na escolha do primeiro elemento como pivô o que pode também aumentar as chances de uma escolha mal feita e residir em um algoritmo de ordenação menos eficiente, vejamos o algoritmo:

A screenshot of a code editor with a dark background and light-colored text. The code is written in Java and implements a QuickSort function using the first element as the pivot. The code is numbered from 1 to 16. The function is named QuickSortFirstPivot and takes two parameters: 'esq' (left) and 'dir' (right). It initializes 'i' to 'esq' and 'j' to 'dir'. It then selects the first element 'array[esq]' as the pivot. A while loop runs as long as 'i' is less than or equal to 'j'. Inside this loop, there are two inner while loops: one that increments 'i' while 'array[i]' is less than the pivot, and another that decrements 'j' while 'array[j]' is greater than the pivot. After these loops, if 'i' is still less than or equal to 'j', the elements at 'i' and 'j' are swapped, and both 'i' and 'j' are incremented/decremented respectively. Finally, the function recursively calls itself on the left and right sub-arrays.

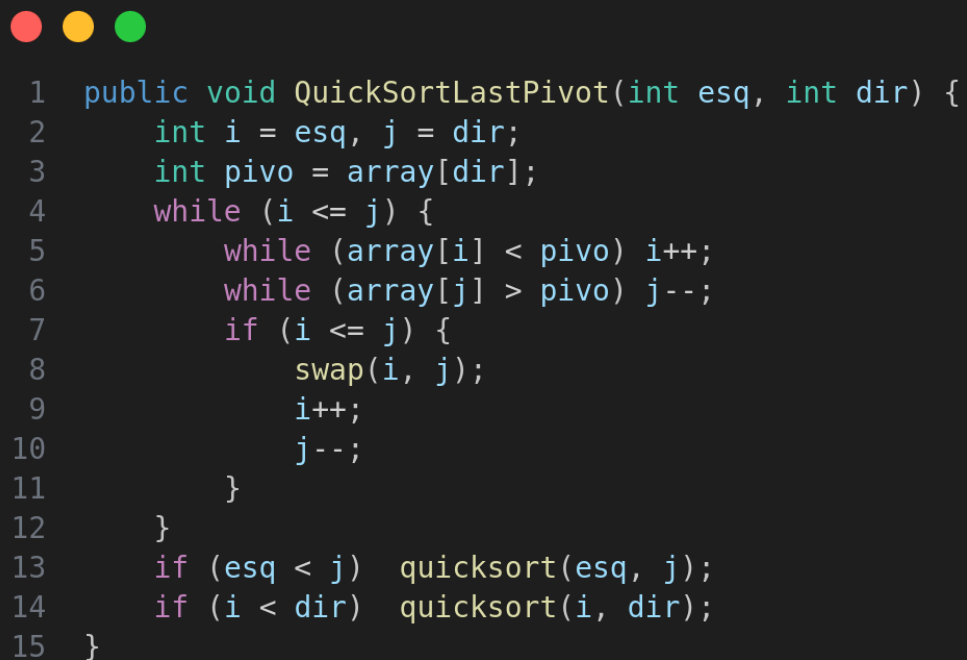
```
1 public void QuickSortFirstPivot(int esq, int dir) {  
2     int i = esq, j = dir;  
3     int pivo = array[esq];  
4     while (i <= j) {  
5         while (array[i] < pivo) i++;  
6         while (array[j] > pivo) j--;  
7         if (i <= j) {  
8             swap(i, j);  
9             i++;  
10            j--;  
11        }  
12    }  
13    if (esq < j) quicksort(esq, j);  
14    if (i < dir) quicksort(i, dir);  
15 }  
16
```

Método de ordenação QuickSortLastPivot - imagem 2

Podemos perceber que a única alteração implementada é a troca do pivô que agora em vez de escolher sempre o elemento na posição ao meio da parcela a se ordenar elemento passou a escolher o elemento mais à esquerda, ou seja o primeiro.

Escolha da última posição do array como pivô - QuickSortLastPivot();

De forma semelhante ao **QuickSortLastPivot** o **QuickSortLastPivot** se baseia na escolha do último elemento como pivô, podendo também aumentar as chances de uma escolha mal feita, vejamos o algoritmo:

A screenshot of a code editor with a dark background and light-colored text. The code is written in Java and implements the QuickSortLastPivot method. It includes line numbers from 1 to 15 on the left side. The code uses standard Java syntax with curly braces for blocks, semicolons for statements, and keywords like public, void, int, while, if, swap, and quicksort. The pivot is chosen as the last element of the array (array[dir]).

```
1 public void QuickSortLastPivot(int esq, int dir) {  
2     int i = esq, j = dir;  
3     int pivo = array[dir];  
4     while (i <= j) {  
5         while (array[i] < pivo) i++;  
6         while (array[j] > pivo) j--;  
7         if (i <= j) {  
8             swap(i, j);  
9             i++;  
10            j--;  
11        }  
12    }  
13    if (esq < j) quicksort(esq, j);  
14    if (i < dir) quicksort(i, dir);  
15 }
```

Método de ordenação QuickSortLastPivot - imagem 3

Podemos perceber novamente que a única alteração implementada é a troca do pivô que agora passou a escolher o elemento mais à direita, ou seja o último.

Escolha de uma posição aleatória é válida dentro do array - QuickSortRandomPivot();

O **QuickSortRandomPivot** se baseia na escolha do pivô a partir de um elemento em uma posição aleatória e que seja válida dentro do array o que a pesar de diminuir um pouco as chances de uma escolha mal feita ainda poderá acarretar em um algoritmo de ordenação menos eficiente, vejamos o algoritmo:

```
1 public void QuickSortRandomPivot(int esq, int dir) {
2     int i = esq, j = dir;
3
4     Random rand = new Random();
5
6     int pivo = array[ Math.abs(esq + rand.nextInt()) % dir];
7     while (i <= j) {
8         while (array[i] < pivo) i++;
9         while (array[j] > pivo) j--;
10        if (i <= j) {
11            swap(i, j);
12            i++;
13            j--;
14        }
15    }
16    if (esq < j) quicksort(esq, j);
17    if (i < dir) quicksort(i, dir);
18 }
```

Método de ordenação QuickSortRandomPivot - imagem 4

Vejamos que as alterações vão desde a criação de um objeto Random dentro do método de QuickSortRandomPivot a uma implementação que possibilite uma escolha de um pivô aleatório conforme um método Math.abs, que impossibilita a presença de números negativo, e mais algumas restrições também feitas com a soma do valor de esq, que mantém a posição do nº aleatório gerado sempre dentro da faixa a se ordenar e por fim tirando o seu resto através do “% dir” que também mantém a posição do nº aleatório sempre dentro da faixa operável no array e conforme a parcela a se ordenar.

E a mediana entre o primeiro elemento, o elemento central e o último elemento - QuickSortMedianOfThree ();

O **QuickSortMedianOfThree** se baseia na escolha do pivô a partir da mediana entre o valor do elemento contido na primeira posição, última posição e com valor do elemento na posição central do array, o que diminui drasticamente a possibilidade de uma escolha mal feita tornando o algoritmo mais eficiente se comparado em grandes escalas, vejamos a sua implementação:

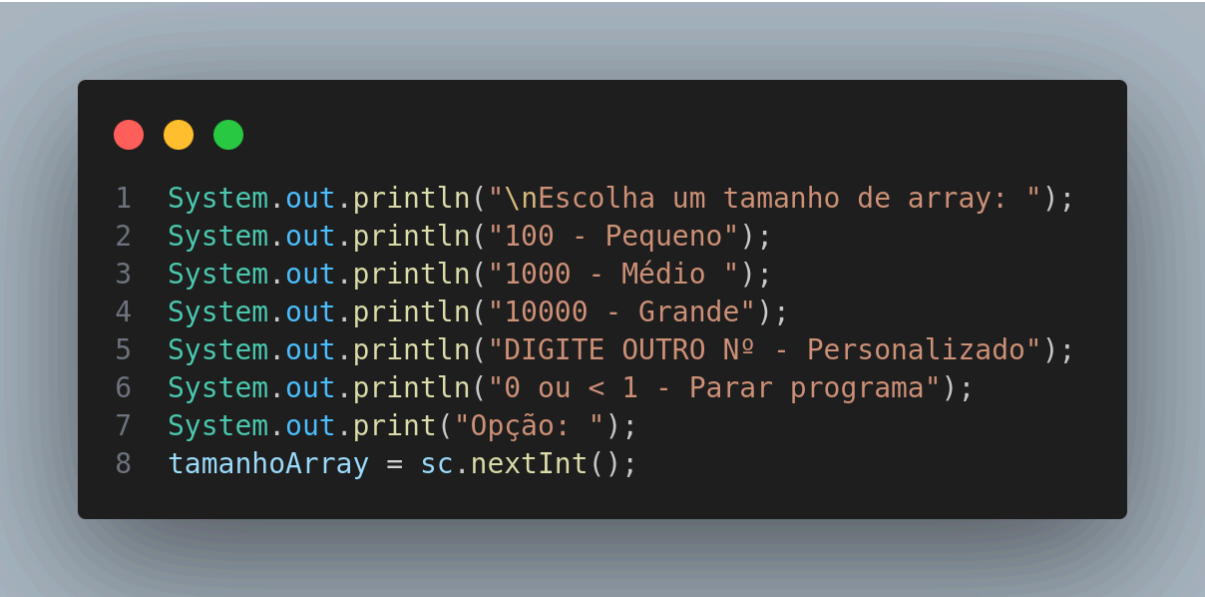
```
1  public void QuickSortMedianOfThree(int esq, int dir) {
2
3      int i = esq, j = dir;
4
5      if (i <= j) {
6
7          int meio = (esq + dir) / 2;
8
9          if (array[esq] > array[meio]) swap(esq, meio);
10         if (array[esq] > array[dir]) swap(esq, dir);
11         if (array[meio] > array[dir]) swap(meio, dir);
12
13         int pivo = array[meio];
14
15         while (i <= j) {
16             while (array[i] < pivo) i++;
17             while (array[j] > pivo) j--;
18             if (i <= j) {
19                 swap(i, j);
20                 i++;
21                 j--;
22             }
23         }
24
25         if (esq < j) QuickSortMedianOfThree(esq, j);
26         if (i < dir) QuickSortMedianOfThree(i, dir);
27     }
28 }
```

Método de ordenação QuickSortMedianOfThree - imagem 5

Após a análise de sua implementação podemos perceber que as alterações implementadas são praticamente um cálculo de mediana, antes da escolha do pivô, conforme explicado anteriormente, para esse cálculo foi implementado primeiramente a comparação do elemento mais à esquerda com o da posição central, caso o da esquerda seja maior, fazemos um swap, ou seja a troca dos elementos contidos em suas posições, e de forma semelhante, compara-se novamente o elemento da esquerda com o mais à direita e posteriormente o do meio com o da direita, fazendo as trocas necessárias, sendo o elemento de pivô sempre o contido na posição central a parcela a se ordenar, ou seja a mediana do elementos analisados.

Tópico 1 - Planejamento de algoritmo de teste, método main();

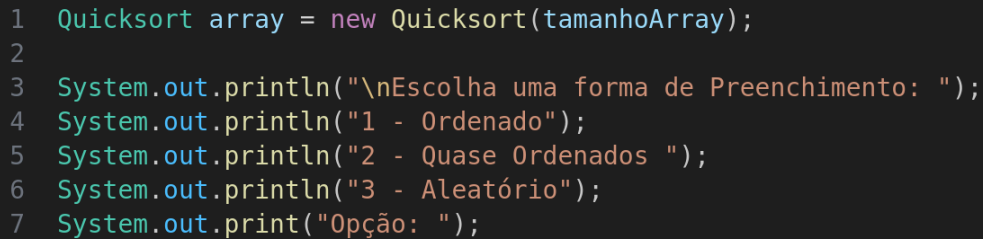
Primeiramente, como entrada temos a definição do tamanho do array a se analisar que tem como opções 100, 1000 10.000 ou um tamanho personalizado, caso entre um nº menor que 1 o programa é abortado, conforme demonstrado abaixo:



```
1 System.out.println("\nEscolha um tamanho de array: ");
2 System.out.println("100 - Pequeno");
3 System.out.println("1000 - Médio ");
4 System.out.println("10000 - Grande");
5 System.out.println("DIGITE OUTRO Nº - Personalizado");
6 System.out.println("0 ou < 1 - Parar programa");
7 System.out.print("Opção: ");
8 tamanhoArray = sc.nextInt();
```

Definição do tamanho do array - imagem 6

Posteriormente, teremos a criação do objeto de Quicksort que nos permitirá operar sobre um array conforme os métodos implementados, e a definição de como esse array criado será preenchido, seja ordenado, quase ordenado ou de forma aleatória:



```
1 Quicksort array = new Quicksort(tamanhoArray);
2
3 System.out.println("\nEscolha uma forma de Preenchimento: ");
4 System.out.println("1 - Ordenado");
5 System.out.println("2 - Quase Ordenados ");
6 System.out.println("3 - Aleatório");
7 System.out.print("Opção: ");
```

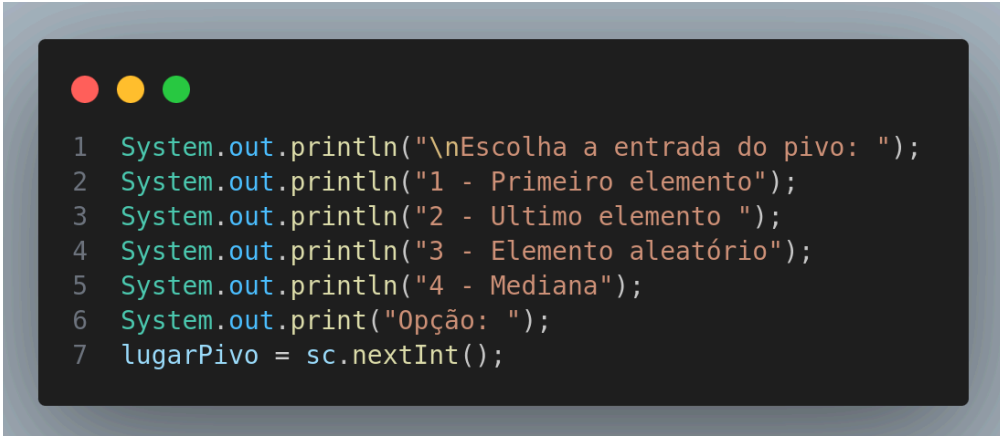
Definição do preenchimento do array - imagem 7



```
1
2 if(formaPreench == 1) array.crescente();
3 else if(formaPreench == 2){ array.crescente();
4     Random rand = new Random();
5
6     for(int i = 0; i < tamanhoArray / 10; i++){
7         array.swap((Math.abs(rand.nextInt()) % (tamanhoArray-1)), (Math.abs(rand.nextInt()) % (tamanhoArray-1)));
8     }
9 }
10 else if(formaPreench == 3) array.aleatorio();
11 else { System.out.println("\nOpção inválida"); break;}
```

Condicionais de escolha de métodos - imagem 8

E por fim a definição do método de ordenação, seja pelo Primeiro elemento - QuickSortFirstPivot (); , Último elemento - QuickSortLastPivot(); , Pivo aleatório - QuickSortRandomPivot(); , Mediana de três elementos (Início, meio e fim) - QuickSortMedianOfThree ();



```
1 System.out.println("\nEscolha a entrada do pivo: ");
2 System.out.println("1 - Primeiro elemento");
3 System.out.println("2 - Ultimo elemento ");
4 System.out.println("3 - Elemento aleatório");
5 System.out.println("4 - Mediana");
6 System.out.print("Opção: ");
7 lugarPivo = sc.nextInt();
```

Definição do pivô para o método de QuickSort - imagem 9

Utilizou-se uma implementação condicional para execução do método escolhido, contabilizando em si, o tempo de execução em segundos e nanosegundo, para a execução de cada e posteriormente após a ordenação um método para a verificação da veracidade da ordenação:

```
Escolha um tamanho de array:
100 - Pequeno
1000 - Médio
10000 - Grande
DIGITE OUTRO Nº - Personalizado
0 ou < 1 - Parar programa
Opção: 10

0 tamanho escolhido foi 10

Escolha uma forma de Preenchimento:
1 - Ordenado
2 - Quase Ordenados
3 - Aleatório
Opção: 3

Escolha a entrada do pivo:
1 - Primeiro elemento
2 - Ultimo elemento
3 - Elemento aleatório
4 - Mediana
Opção: 3

Array antes:
[ (0)5 (1)0 (2)2 (3)4 (4)1 (5)8 (6)7 (7)6 (8)3 (9)9]

Array depois:
[ (0)0 (1)1 (2)2 (3)3 (4)4 (5)5 (6)6 (7)7 (8)8 (9)9]

0 tamanho escolhido foi 10

Tempo para ordenar: 0.0 s
Tempo para ordenar: 34283.0 nanosegundos


isOrdenado: true
```

Exemplo de execução no terminal com array tamanho 10 - imagem 10

Tópico 2 - O desempenho observado em cada cenário, apresentado o gráfico com o tempo de execução

Nesta seção será feita uma análise técnica sobre o desempenho de cada algoritmo de ordenação a partir de tabelas feitas com a coleta de dados de execução em cada um dos testes feitos com cada método de ordenação, demonstrando de forma prática as discussões anteriormente feitas.

Os teste foram feitos primeiramente com 3 variações de tamanhos de array, sendo eles 100, 1000, 10.000, também variando em tipos diferentes de preenchimento do array, sendo eles ordenados, quase ordenados no qual foi padronizado uma faixa de 10% de desordenação e um array preenchido com elementos de valores aleatórios.

A screenshot of a code editor window with a dark background and light-colored text. The code is in Java and implements a loop that iterates over the first 10% of an array, swapping each element with a random element from the rest of the array to create a 10% unsorted state. The code is as follows:

```
1 for(int i = 0; i < tamanhoArray / 10; i++){
2     array.swap((Math.abs(rand.nextInt()) % (tamanhoArray-1)), (Math.abs(rand.nextInt()) % (tamanhoArray-1)));
3 }
```

Método para definição da faixa de 10% de desordenação - imagem 11

Escolha da primeira posição do array como pivô - QuickSortLastPivot();

TABELA 1		
Pivô:	Primeiro elemento	
Método:	QuickSortFirstPivot ();	
Ordenação:	Ordenados	
Qts elementos	Tempo em segundos	Tempo em nanosegundos
100	0	16771
1000	0,001	90917
10000	0,001	1456061
Ordenação:	Quase ordenados	
Qts elementos	Tempo em segundos	Tempo em nanosegundos
100	0	13094
1000	0	81980
10000	0,001	1130540
Ordenação:	Aleatórios	
Qts elementos	Tempo em segundos	Tempo em nanosegundos
100	0	19776
1000	0	212491
10000	0.003 s	2734467

Tabela completa escolha do primeiro elemento, método QuickSortFirstPivot - Tabela 1

Primeiro elemento - QuickSortFirstPivot ();			
Qts Elementos	Ordenados (nanosegundos)	Quase ordenados (nanosegundos)	Aleatórios (nanosegundos)
100	16771	13094	19776
1000	90917	81980	212491
10000	1456061	1130540	2734467

Tabela reduzida escolha do primeiro elemento, método QuickSortFirstPivot - Tabela 1.1

Ordenados (nanosegundos), Quase ordenados (nanosegundos) e Aleatórios (nanosegundos)

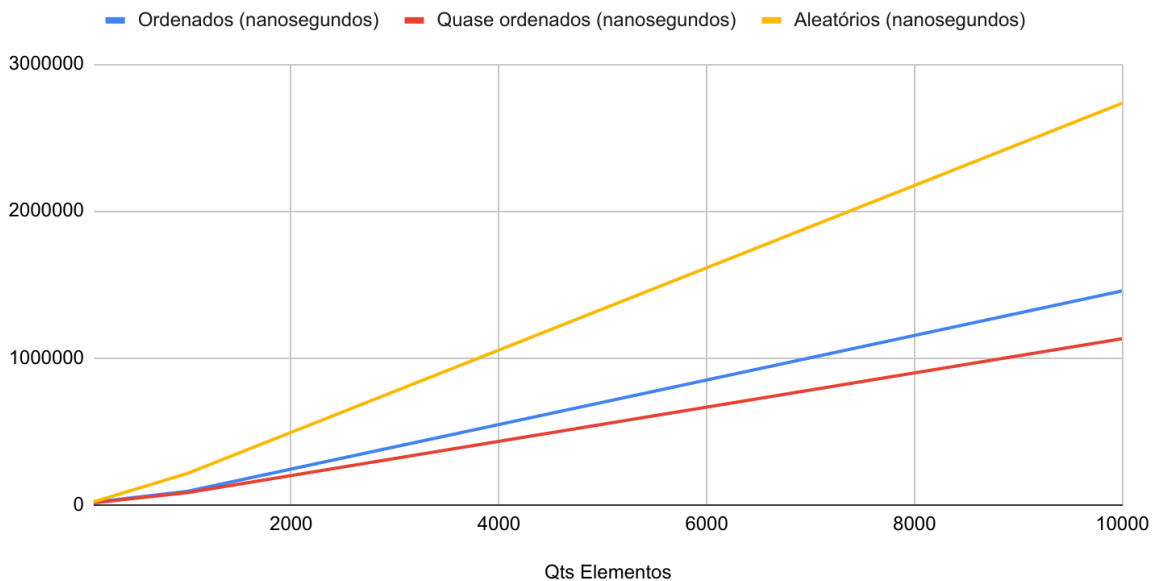


Gráfico escolha do primeiro elemento, método QuickSortFirstPivot - Gráfico 1

Escolha da última posição do array como pivô - QuickSortLastPivot();

TABELA 2		
Pivô:	Ultimo elemento	
Metodo:	QuickSortLastPivot();	
Ordenação:	Ordenados	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	64418
1000	0	52677
10000	0,001	464504
Ordenação:	Quase ordenados	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	60611
1000	0	460587
10000	0,001	1166849
Ordenação:	Aleatórios	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	67391
1000	0,001	184943
10000	0,003	2449186

Tabela completa escolha do último elemento, método QuickSortLastPivot - Tabela 2

Ultimo elemento - QuickSortLastPivot();			
Qts Elementos	Ordenados (nanosegundos)	Quase ordenados (nanosegundos)	Aleatórios (nanosegundos)
100	64418	60611	67391
1000	52677	460587	184943
10000	464504	1166849	2449186

Tabela reduzida escolha do último elemento, método QuickSortLastPivot - Tabela 2.1

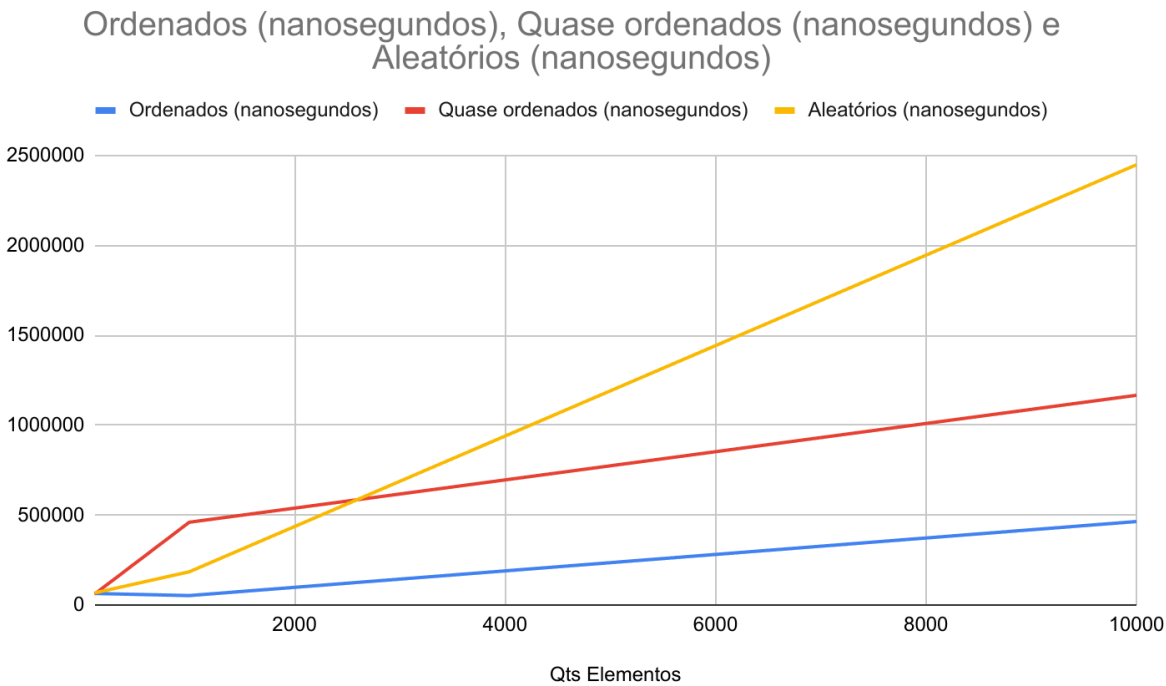


Gráfico escolha do último elemento, método QuickSortLastPivot - Gráfico 2

**Escolha de uma posição aleatória é válida dentro do array -
QuickSortRandomPivot();**

TABELA 3		
Pivô:	Pivo aleatório	
Método:	QuickSortRandomPivot();	
Ordenação:	Ordenados	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	49349
1000	0	61982
10000	0	336830
Ordenação:	Quase ordenados	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	19135
1000	0	96245
10000	0,001	781145
Ordenação:	Aleatórios	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	21329
1000	0	200586
10000	0,005	4942917

Tabela completa escolha de elemento aleatório, método QuickSortRandomPivot - Tabela 3

Pivo aleatório - QuickSortRandomPivot();			
Qts Elementos	Ordenados (nanosegundos)	Quase ordenados (nanosegundos)	Aleatórios (nanosegundos)
100	49349	19135	21329
1000	61982	96245	200586
10000	336830	781145	4942917

Tabela reduzida escolha de elemento aleatório, método QuickSortRandomPivot - Tabela 3.1

Ordenados (nanosegundos), Quase ordenados (nanosegundos) e Aleatórios (nanosegundos)

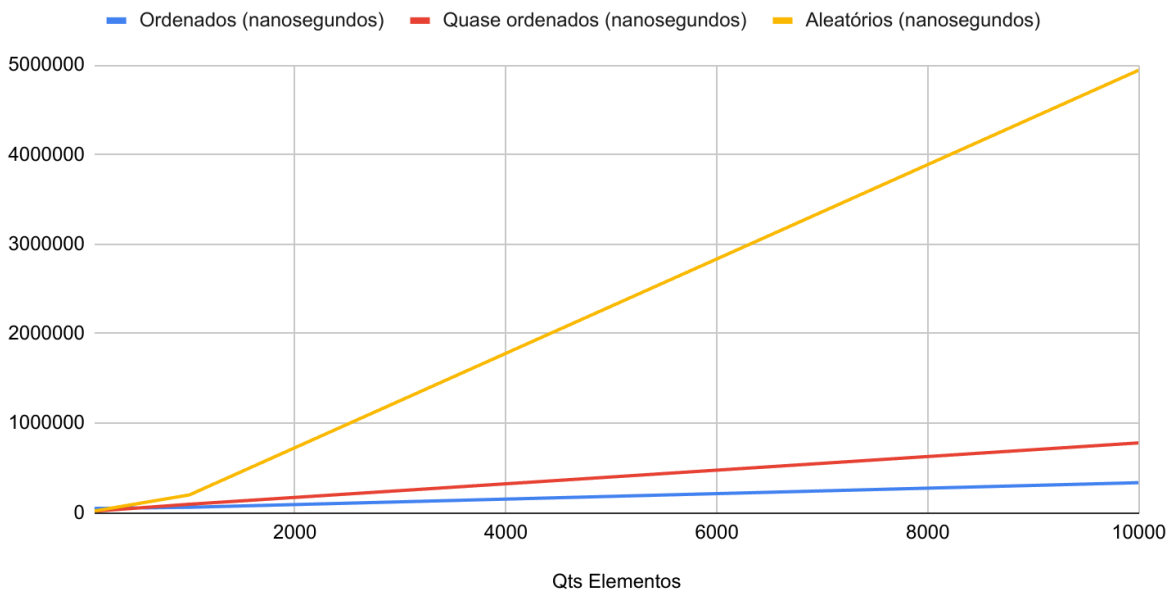


Gráfico escolha de elemento aleatório, método QuickSortRandomPivot - Gráfico 3

E a mediana entre o primeiro elemento, o elemento central e o último elemento - QuickSortMedianOfThree ();

TABELA 4		
Pivô:	Mediana de três elementos (Início, meio e fim)	
Método:	QuickSortMedianOfThree ();	
Ordenação	Ordenados	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	22441
1000	0	51143
10000	0,001	311088
Ordenação:	Quase ordenados	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	54470
1000	0,001	535307
10000	0,002	1322601
Ordenação:	Aleatórios	
Qts elementos	Tempo em segundos	Tempo em nanodegundos
100	0	29044
1000	0	262328
10000	0,002	1549616

Tabela completa, mediana dos elementos,, método QuickSortMedianOfThree - Tabela 4

Mediana de três elementos (Início, meio e fim) - QuickSortMedianOfThree ();			
Qts Elementos	Ordenados (nanosegundos)	Quase ordenados (nanosegundos)	Aleatórios (nanosegundos)
100	22441	54470	29044
1000	51143	535307	262328
10000	311088	1322601	1549616

Tabela reduzida, mediana dos elementos,, método QuickSortMedianOfThree - Tabela 4.1

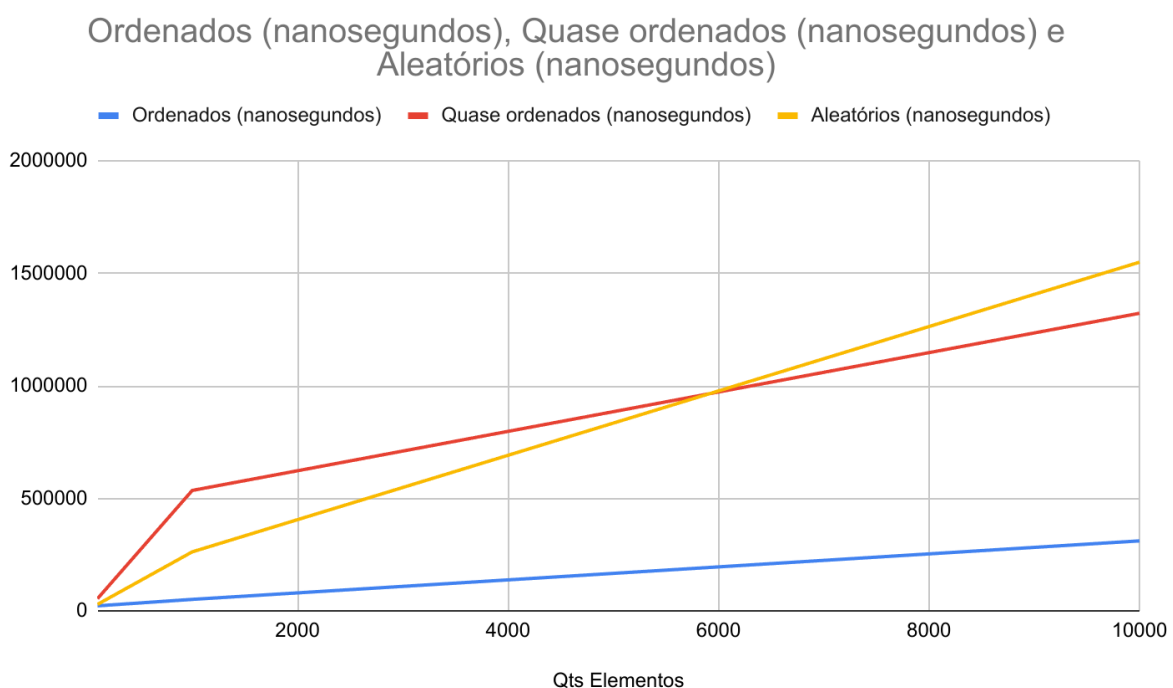


Gráfico mediana dos elementos, método QuickSortMedianOfThree - Gráfico 4

Análise geral e comparação de métodos

Media geral por método	
Método	Tempo máximo (nanosegundo)
QuickSortFirstPivot ();	2734467
QuickSortLastPivot();	2449186
QuickSortRandomPivot();	4942917
QuickSortMedianOfThree ();	1549616

Gráfico contendo o tempo máximo gasto por cada algoritmo de ordenação - Tabela 5

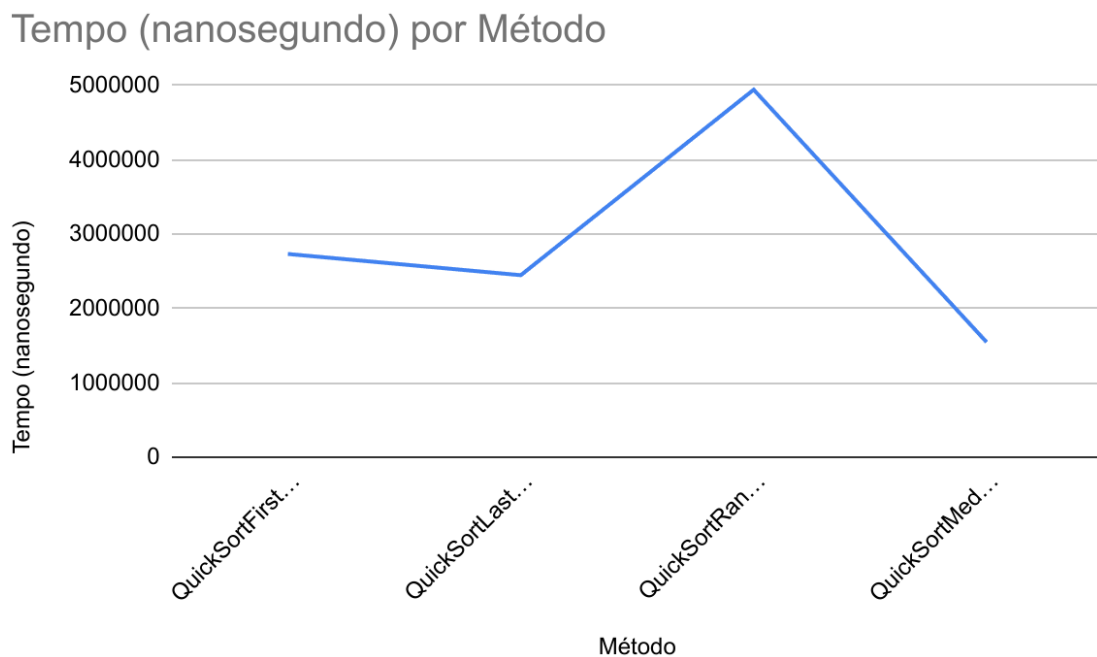


Gráfico contendo a comparação do tempo máximo gasto por cada algoritmo de ordenação - Gráfico 5

Tópico 3 - Uma discussão sobre qual estratégia foi mais eficiente em cada caso e por quê

A partir da análise dos resultados obtidos pode-se observar que para as estrutura de dados já ordenados a maioria dos métodos apresentaram uma certa semelhança de gasto de tempo, haja vista a ordenação já presente no array, com exceção do método QuickSortFirstPivot (); que demonstrou uma certa ineficiência, ou seja um gasto de tempo maior comparado aos demais métodos, para as estrutura de dados quase ordenados o método que se demonstrou mais eficiente foi o QuickSortRandomPivot();, possivelmente devido a uma certa geração de aleatoriedade que o beneficiou nesta ordenação, no entanto acredita-se que essa aleatoriedade não seja a mais eficiente em todos os caso, tendo em vista que há a possibilidade da escolha de um pivô não muito favorável, no entanto para as estrutura de dados aleatórios o método QuickSortMedianOfThree (); apresentou um gasto de tempo consideravelmente menor aos demais métodos principalmente quando o tamanho da estrutura de dados aumentava sua quantidade de elementos, no entanto se comparada sua eficiência com estrutura de dados consideravelmente menores essa diferença é mínima. No entanto, de forma geral o método QuickSortMedianOfThree(); apresentou resultados mais consistentes e com menor impacto ao crescimento do volume dos dados a serem ordenados

Tópico 4 - Links úteis:

Link de acesso às planilhas utilizadas:

<https://docs.google.com/spreadsheets/d/1wgbfh0xICwB8OxzmHx7h5FPpYZHxtr5aNmHkAalZvUl/edit?usp=sharing>

Link de acesso ao meu GitHub completo:

https://github.com/DaviKandido/Algoritimos_e_Estrutura_de_Dados_II

O arquivo da atividade de encontrar em /LABS_Exercicios/lab06:

https://github.com/DaviKandido/Algoritimos_e_Estrutura_de_Dados_II/tree/main/LABS_Exercicios/lab06

O arquivo principal: [VariacaoPivoQuickshort.java](#)