

O funcionamento de cada estratégia de escolha do pivô:

- Primeiro Elemento:

O algoritmo utiliza o primeiro elemento do array como pivô. Esta é uma estratégia simples, mas nem sempre a mais eficiente. Dependendo da natureza dos dados, pode levar a uma pior performance.

```
41  public static void QuickSortFirstPivot(int esq, int dir, int[] array){
42      int i = esq, j = dir;
43      int pivo = array[esq];
44
45      while (i <= j){
46          while(array[i] < pivo){
47              i++;
48          }
49          while(array[j] > pivo){
50              j--;
51          }
52
53          if(i <= j){
54              swap(i, j, array);
55              i++;
56              j--;
57          }
58      }
59
60      if(esq < j){
61          QuickSortFirstPivot(esq, j, array);
62      }
63      if(i < dir){
64          QuickSortFirstPivot(i, dir, array);
65      }
66  }
```

- Último Elemento:

o pivô sendo o último elemento do array é uma escolha válida, mas não necessariamente a mais eficiente para todos os tipos de dados. Ela pode levar a problemas de performance no caso de arrays ordenados.

```

68  public static void QuickSortLastPivot(int esq, int dir, int[] array){
69      int i = esq, j = dir;
70      int pivo = array[dir];
71
72      while (i <= j){
73          while(array[i] < pivo){
74              i++;
75          }
76          while(array[j] > pivo){
77              j--;
78          }
79
80          if(i <= j){
81              swap(i, j, array);
82              i++;
83              j--;
84          }
85      }
86
87      if(esq < j){
88          QuickSortLastPivot(esq, j, array);
89      }
90      if(i < dir){
91          QuickSortLastPivot(i, dir, array);
92      }
93  }

```

- Pivô Aleatório:

Escolher um pivô aleatoriamente é uma estratégia poderosa que minimiza o risco de pior caso e torna o algoritmo mais robusto, especialmente contra inputs adversários.

```

95  public static void QuickSortRandomPivot(int esq, int dir, int[] array){
96      int i = esq, j = dir;
97      int pivo = array[(dir + esq) / 2];
98
99      while (i <= j){
100          while(array[i] < pivo){
101              i++;
102          }
103          while(array[j] > pivo){
104              j--;
105          }
106
107          if(i <= j){
108              swap(i, j, array);
109              i++;
110              j--;
111          }
112      }
113
114      if(esq < j){
115          QuickSortRandomPivot(esq, j, array);
116      }
117      if(i < dir){
118          QuickSortRandomPivot(i, dir, array);
119      }
120  }

```

- Mediana de Três Elementos:

A estratégia de Mediana de Três é uma forma eficaz de escolher um pivô que tende a gerar partições mais equilibradas, melhorando a eficiência do algoritmo.

```

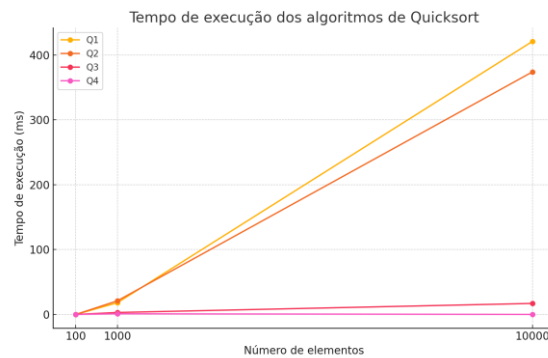
122     public static void QuickSortMedianOfThree(int esq, int dir, int[] array){
123         int i = esq, j = dir;
124         int tmp = (esq + dir) / 2;
125
126         if(array[esq] > array[tmp]){
127             swap(esq, tmp, array);
128         }
129         if(array[esq] > array[dir]){
130             swap(esq, dir, array);
131         }
132         if(array[tmp] > array[dir]){
133             swap(tmp, dir, array);
134         }
135
136         int pivo = array[tmp];
137         while (i <= j){
138             while(array[i] < pivo){
139                 i++;
140             }
141             while(array[j] > pivo){
142                 j--;
143             }
144
145             if(i <= j){
146                 swap(i, j, array);
147                 i++;
148                 j--;
149             }
150         }
151
152         if(esq < j){
153             QuickSortMedianOfThree(esq, j, array);
154         }
155         if(i < dir){
156             QuickSortMedianOfThree(i, dir, array);
157         }
158     }

```

Desempenho representado com gráficos e tabelas:

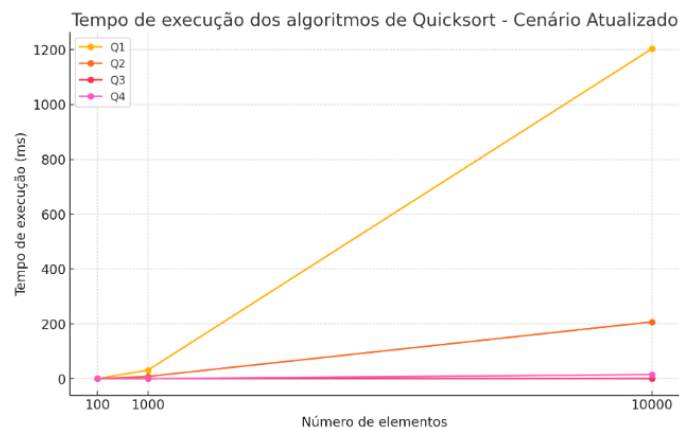
- Cenário 1 – Array Ordenado Crescente:

	-	100	1000	10.000
Q1		0.0	18.0	421.0
Q2		0.0	21.0	374.0
Q3		0.0	3.0	17.0
Q4		0.0	1.0	0.0



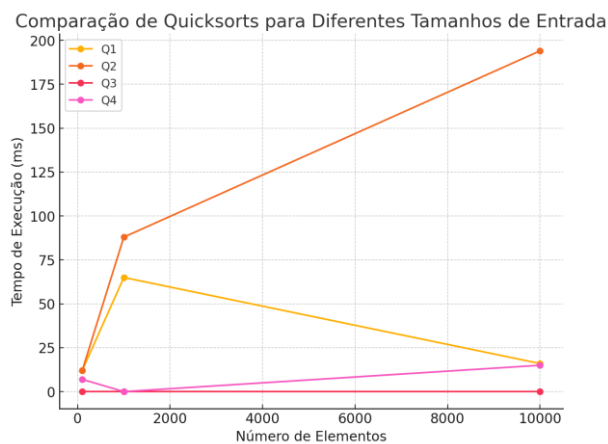
- Cenário 2 – Array Ordenado Decrescente:

	-	100	1000	10.000
Q1		0.0	31.0	1203.0
Q2		0.0	8.0	207.0
Q3		0.0	0.0	0.0
Q4		0.0	0.0	15.0



- Cenário 3 – Array Parcial Ordenado Crescente:

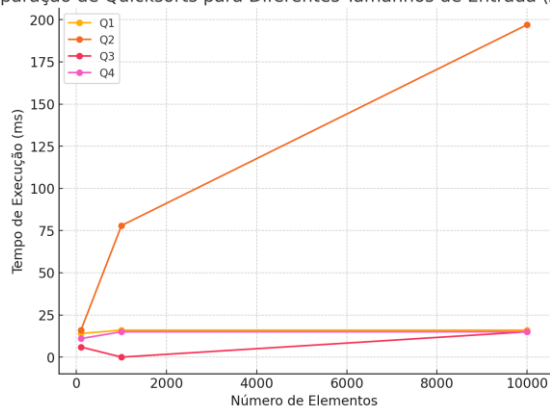
	-	100	1000	10.000
Q1		12.0	65.0	16.0
Q2		12.0	88.0	194.0
Q3		0.0	0.0	0.0
Q4		7.0	0.0	15.0



- Cenário 4 – Array Parcial Ordenado Decrescente:

	-	100	1000	10.000
Q1		14.0	16.0	16.0
Q2		16.0	78.0	197.0
Q3		6.0	0.0	15.0
Q4		11.0	15.0	15.0

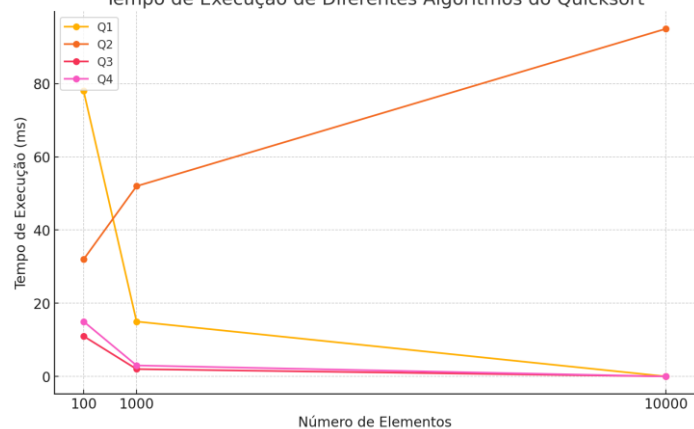
Comparação de Quicksorts para Diferentes Tamanhos de Entrada (Atualizado)



- Cenário 5 – Array Aleatório:

	-	100	1000	10.000
Q1		78.0	15.0	0.0
Q2		32.0	52.0	95.0
Q3		11.0	2.0	0.0
Q4		15.0	3.0	0.0

Tempo de Execução de Diferentes Algoritmos do Quicksort



Discussão sobre qual estratégia é mais eficiente e porquê:

A mediana de três é a mais eficiente na maioria dos casos porque garante que o pivô seja representativo dos dados. Isso leva a partições mais balanceadas e reduz a chance de o algoritmo precisar fazer muitas chamadas recursivas.