PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Curso Superior de Ciência da Computação

Algoritmos e Estruturas de Dados II:
QuickSort e Seu Pivo

Domynic Barros Lima

Belo Horizonte
2024

SUMÁRIO

1 - ENTENDENDO O PROBLEMA

O QuickSort é um algoritmo de ordenação eficiente que opera de maneira recursiva. Em sua execução, escolhemos um elemento chamado "pivô", que serve como referência para a comparação e organização dos elementos em um vetor. Durante sua execução, dois ponteiros, "I" e "J", são criados e percorrem o vetor em busca do primeiro elemento que seja menor e maior que o pivô. Quando esses elementos são encontrados, eles são trocados de posição (swap). Esse procedimento continua até que os ponteiros se cruzem. Ao final dessa primeira passagem, garantimos que todos os elementos menores que o pivô estão de um lado e todos os elementos maiores estão do outro, resultando em um vetor mais organizado (semi-ordenado).

A partir desse ponto, o vetor é dividido em dois subvetores, e o algoritmo é reexecutado para cada um deles até que estejam completamente ordenados.

A escolha do pivô no QuickSort é crucial para o desempenho do algoritmo, pois influencia diretamente a eficiência da ordenação. Um pivô bem selecionado pode dividir o vetor de maneira equilibrada, resultando em subvetores de tamanhos semelhantes e, assim, reduzindo a profundidade da recursão, apresentando um tempo de execução próximo a $O(n * \log(n))$ no melhor caso. Em contrapartida, uma escolha inadequada pode levar a divisões desbalanceadas, em que um dos subvetores é muito maior que o outro. Isso faz com que o algoritmo se comporte de forma ineficiente, apresentando um tempo de execução próximo a $O(n^2)$ no pior caso.

Mas como podemos garantir uma boa escolha do pivô? Essa é uma pergunta complexa, pois a melhor escolha depende do tamanho do vetor e da disposição dos elementos. Como não temos certeza sobre esses fatores, geralmente escolhemos o pivô como sendo o elemento central do vetor.

Neste trabalho, iremos analisar algumas estratégias para a escolha do pivô, incluindo a mediana de três elementos, a seleção de um pivô aleatório e a escolha do primeiro ou último elemento do vetor. Em seguida, avaliaremos os resultados para compreender melhor o comportamento do algoritmo.

2 – ESTRATÉGIAS NA ESCOLHA DO PIVÔ

2.1 - QuickSort Padrão com Pivô Central

A primeira estratégia que analisaremos é o QuickSort padrão, onde o pivô é escolhido como o elemento central do vetor. Essa abordagem tem a vantagem de, em muitos casos, proporcionar uma divisão equilibrada dos elementos. Ao selecionar o pivô central, é mais provável que ele esteja próximo da mediana dos valores, o que ajuda a garantir que os subvetores resultantes tenham tamanhos semelhantes.

```java
// Pivô como elemento do meio (Padrão)
public void quicksort(int[] array, int esq, int dir) {
    int i = esq, j = dir;
    int pivo = array[(dir + esq) / 2];
    while (i <= j) {
        while (array[i] < pivo) { comparacoes++; i++; }
        while (array[j] > pivo) { comparacoes++; j--; }
        if (i <= j) {
            swap(array, i, j);
            i++;
            j--;
        }
    }
    if (esq < j) quicksort(array, esq, j);
    if (i < dir) quicksort(array, i, dir);
}
```

Vantagens:

- **Divisão Equilibrada:** Ao escolher o elemento central, a divisão dos elementos em subvetores tende a ser mais equilibrada, resultando em profundidades de recursão menores. Isso, por sua vez, pode levar a um desempenho mais eficiente em comparação com outras estratégias, especialmente em listas de tamanho médio.
- **Menos Chamadas Recursivas:** Uma boa escolha de pivô diminui o número de chamadas recursivas necessárias para ordenar os subvetores, o que pode reduzir o tempo de execução total do algoritmo.

Desvantagens:

- Desempenho em Casos Especiais: **Em situações em que os dados estão ordenados ou quase ordenados, escolher o pivô central pode não ser tão eficiente, pois ainda pode levar a um desempenho próximo de $O(n^2)$. Nesse caso, o algoritmo pode se comportar de maneira semelhante ao uso do primeiro ou do último elemento como pivô.**

Essa estratégia é frequentemente um bom ponto de partida para a implementação do QuickSort, mas é importante considerar outras opções em cenários em que a distribuição dos dados não é aleatória.

2.2 - QuickSort com Pivô como Primeiro Elemento

A segunda estratégia que iremos explorar é o QuickSort que utiliza o primeiro elemento do vetor como o pivô. Essa abordagem é simples de implementar, mas suas implicações em termos de desempenho podem variar significativamente dependendo da distribuição dos dados.

```java
// Pivô como primeiro elemento
public void quicksortFirstPivot(int[] array, int esq, int dir) {
    if (esq >= dir) return;
    int i = esq, j = dir;
    int pivo = array[esq];
    while (i <= j) {
        while (array[i] < pivo) { comparacoes++; i++; }
        while (array[j] > pivo) { comparacoes++; j--; }
        if (i <= j) {
            swap(array, i, j);
            i++;
            j--;
        }
    }
    if (esq < j) quicksortFirstPivot(array, esq, j);
    if (i < dir) quicksortFirstPivot(array, i, dir);
}
```

Vantagens:

- **Implementação Simples:** A escolha do primeiro elemento como pivô facilita a implementação do algoritmo, uma vez que não é necessário calcular ou identificar um elemento específico, como a mediana ou um elemento central. Isso pode ser vantajoso em cenários onde a simplicidade do código é priorizada.

Desvantagens:

- **Desempenho em Dados Ordenados:** Uma das maiores desvantagens dessa estratégia é seu desempenho em casos onde o vetor está ordenado ou quase ordenado. Nesse cenário, a escolha do primeiro elemento como pivô resulta em subvetores altamente desbalanceados. O algoritmo pode se comportar como $O(n^2)$ em tais situações, devido à profundidade da recursão aumentada, levando a muitas chamadas recursivas.

- **Divisões Desbalanceadas:** Ao usar o primeiro elemento, a divisão entre os subvetores tende a ser desigual, o que compromete a eficiência do algoritmo. Quando um subarray se torna significativamente maior que o outro, o tempo de execução do QuickSort pode aumentar substancialmente.

Devido a essas desvantagens, a escolha do primeiro elemento como pivô é geralmente menos recomendada em implementações de QuickSort que necessitam de um desempenho consistente, especialmente em listas que podem não ter uma distribuição aleatória. Essa estratégia pode ser útil em casos específicos, mas é fundamental estar ciente de suas limitações.

2.3 - QuickSort com Pivô como Último Elemento

A terceira estratégia que analisaremos é a utilização do último elemento do vetor como o pivô. Essa abordagem, assim como a escolha do primeiro elemento, é fácil de implementar e pode ser útil em certas circunstâncias.

```java
// Pivô como último elemento
public void quicksortLastPivot(int[] array, int esq, int dir) {
    if (esq >= dir) return;
    int i = esq, j = dir;
    int pivo = array[dir];
    while (i <= j) {
        while (array[i] < pivo) { comparacoes++; i++; }
        while (array[j] > pivo) { comparacoes++; j--; }
        if (i <= j) {
            swap(array, i, j);
            i++;
            j--;
        }
    }
    if (esq < j) quicksortLastPivot(array, esq, j);
    if (i < dir) quicksortLastPivot(array, i, dir);
}
```

Vantagens:

- **Implementação Simples:** A escolha do último elemento como pivô simplifica a implementação do algoritmo, pois não requer cálculos adicionais para determinar o pivô. O último elemento pode ser facilmente acessado, tornando o código mais claro e direto.

- **Consistência em Alguns Cenários:** Em situações em que o vetor apresenta uma ordem aleatória ou uma distribuição relativamente equilibrada, o uso do último elemento pode resultar em divisões que são razoavelmente balanceadas, permitindo que o algoritmo opere de forma eficiente.

Desvantagens:

- **Desempenho em Dados Ordenados:** Similar à estratégia de escolher o primeiro elemento como pivô, usar o último elemento pode ser desastroso em listas que já estão ordenadas ou quase ordenadas. Nesse caso, o algoritmo pode se comportar como $O(n^2)$, resultando em um número excessivo de chamadas recursivas e aumentando o tempo de execução.
- **Divisões Desbalanceadas:** Como a escolha do último elemento não leva em conta a distribuição geral dos dados, é comum que ocorram divisões desbalanceadas. Isso pode levar a um desempenho inferior em comparação com estratégias que buscam um pivô mais centralizado ou mediano.

Portanto, embora a escolha do último elemento como pivô seja uma opção simples e direta, ela também apresenta limitações significativas em termos de desempenho. Essa estratégia pode ser viável em certos contextos, mas, assim como na escolha do primeiro elemento, é importante avaliar a distribuição dos dados antes de sua aplicação.

2.4 - QuickSort com Pivô Aleatório

A quarta estratégia que iremos explorar é a escolha do pivô aleatório. Nesse método, um elemento é selecionado aleatoriamente do vetor para servir como pivô em cada chamada recursiva. Essa abordagem visa minimizar os efeitos negativos de distribuições desfavoráveis de dados.

```java
// Pivô aleatório
public void quicksortRandomPivot(int[] array, int esq, int dir) {
    if (esq >= dir) return;
    Random rand = new Random();
    int i = esq, j = dir;
    int pivoIndex = esq + rand.nextInt(dir - esq + 1);
    int pivo = array[pivoIndex];

    while (i <= j) {
        while (array[i] < pivo) { comparacoes++; i++; }
        while (array[j] > pivo) { comparacoes++; j--; }
        if (i <= j) {
            swap(array, i, j);
            i++;
            j--;
        }
    }
    if (esq < j) quicksortRandomPivot(array, esq, j);
    if (i < dir) quicksortRandomPivot(array, i, dir);
}
```

Vantagens:

- **Redução de Desempenho Pior Caso:** A principal vantagem da escolha aleatória do pivô é que ela ajuda a evitar os piores casos de desempenho que podem ocorrer com listas já ordenadas ou quase ordenadas. Ao escolher um pivô aleatoriamente, é menos provável que o algoritmo se depare com uma sequência de elementos que causem divisões desbalanceadas.
- **Balanceamento Esperado:** Embora a escolha do pivô aleatório não garanta divisões sempre balanceadas, ela oferece um desempenho esperado de O(n * log(n)) na média, já que a aleatoriedade tende a levar a divisões mais equilibradas ao longo do tempo.

Desvantagens:

- **Overhead de Aleatoriedade:** A implementação de um algoritmo que seleciona um pivô aleatório pode introduzir um pequeno overhead devido à necessidade de gerar um número aleatório. Embora esse custo seja geralmente pequeno, ele pode ser significativo em listas muito grandes ou em situações onde a eficiência máxima é crítica.

- **Possibilidade de Pior Desempenho em Casos Específicos:** Apesar de a escolha aleatória do pivô melhorar o desempenho médio, ainda existe a possibilidade de que uma série de escolhas ruins ocorra em casos específicos, resultando em divisões desbalanceadas. Embora essa situação seja rara, não é impossível.

Em resumo, a estratégia de escolher um pivô aleatório no QuickSort oferece uma abordagem robusta para melhorar o desempenho em diversos casos. Embora possa introduzir algum overhead, suas vantagens em evitar piores cenários a tornam uma escolha atraente, especialmente em implementações que lidam com dados de forma imprevisível.

2.5 - QuickSort com Pivô como Mediana (primeiro, meio, ultimo)

A última estratégia que iremos analisar é a escolha do pivô com base na mediana de três elementos: o primeiro, o meio e o último do vetor. Essa abordagem busca otimizar a escolha do pivô ao garantir que ele seja um valor que esteja mais centralizado em relação aos outros elementos do vetor.

```java
// Pivô como Mediana de três (primeiro, meio, último)
public void quicksortMedianOfThree(int[] array, int esq, int dir) {
    if (esq >= dir) return;
    int i = esq, j = dir;
    int mid = (esq + dir) / 2;
    int pivo = median(array[esq], array[mid], array[dir]);
    while (i <= j) {
        while (array[i] < pivo) { comparacoes++; i++; }
        while (array[j] > pivo) { comparacoes++; j--; }
        if (i <= j) {
            swap(array, i, j);
            i++;
            j--;
        }
    }
    if (esq < j) quicksortMedianOfThree(array, esq, j);
    if (i < dir) quicksortMedianOfThree(array, i, dir);
}

// Função para calcular a mediana de três valores
private int median(int a, int b, int c) {
    if ((a > b) ^ (a > c)) return a;
    else if ((b > a) ^ (b > c)) return b;
    else return c;
}
```

Vantagens:

- **Melhor Equilíbrio nas Divisões:** Ao escolher a mediana dos três elementos (primeiro, meio e último), aumentamos a probabilidade de que o pivô selecionado esteja próximo do valor mediano do vetor. Isso tende a resultar em divisões mais equilibradas, reduzindo a profundidade da recursão e melhorando a eficiência do algoritmo.

- **Redução de Desempenho Pior Caso:** Essa estratégia ajuda a evitar os piores casos de desempenho que podem ocorrer em listas ordenadas ou quase ordenadas. A seleção da mediana aumenta a chance de uma escolha que não cause divisões excessivamente desbalanceadas, mantendo o desempenho mais próximo de O(n log n) em média.

Desvantagens:

- **Custo de Cálculo da Mediana:** O cálculo da mediana dos três elementos requer comparações adicionais, o que pode introduzir um pequeno overhead. Embora essa penalização seja geralmente pequena, ela pode ser um fator a ser considerado em cenários de alto desempenho, especialmente com listas muito grandes.
- **Implementação Adicional:** Essa abordagem requer um pouco mais de complexidade na implementação, uma vez que é necessário adicionar lógica para determinar a mediana dos três elementos. Isso pode tornar o código menos direto em comparação com a escolha do primeiro ou último elemento como pivô.

Em resumo, a estratégia de escolher o pivô como a mediana dos três elementos é uma abordagem eficaz que equilibra a eficiência e a simplicidade. Ao aumentar a probabilidade de divisões balanceadas, essa técnica melhora o desempenho geral do QuickSort, tornando-se uma das opções mais recomendadas para a escolha do pivô em diversas situações.

---

3 – ANÁLISE DE DESMPENHO

Cada um das cinco estrategias foram testadas com vetores de tamanho 100, 1.000, 10.000, 100.000 e 1.000.000. Com os elementos dos vetores estando em quatro condições de ordenação, Ordenação Aleatória, Ordenação Completa Crescente, Ordenação Completa Decrescente e Ordenação Semi Completa. Foram Registrados o Tempo de execução de cada Algoritmo em milisegundos (ms), a quantidade de comparações entre elementos do vetor e a quantidade de movimentações entre elementos do vetor.

3.1 – Tabelas de Dados para Vetores de Tamanho 100

3.1.1 - Ordenação Aleatória

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	0ms	543	183
QuickSort Primeiro	0ms	495	194
QuickSort Último	0ms	492	190
QuickSort Aleatório	0ms	501	181
QuickSort Mediana	0ms	401	191

3.1.2 - Ordenação Completa Crescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	0ms	482	181
QuickSort Primeiro	0ms	532	186
QuickSort Último	0ms	510	185
QuickSort Aleatório	0ms	472	175
QuickSort Mediana	0ms	478	182

3.1.3 - Ordenação Completa Decrescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	0ms	545	174
QuickSort Primeiro	0ms	644	178
QuickSort Último	0ms	673	181
QuickSort Aleatório	0ms	511	176
QuickSort Mediana	0ms	501	180

3.1.4 - Ordenação Semi Completa

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	0ms	382	179
QuickSort Primeiro	0ms	449	190
QuickSort Último	0ms	455	193
QuickSort Aleatório	0ms	484	180
QuickSort Mediana	0ms	378	185

3.2 – Tabelas de Dados para Vetores de Tamanho 1.000

3.2.1 - Ordenação Aleatória

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	1ms	7513	2589
QuickSort Primeiro	0ms	8308	2657
QuickSort Último	1ms	8510	2646
QuickSort Aleatório	1ms	11158	2531
QuickSort Mediana	0ms	6456	2661

3.2.2 - Ordenação Completa Crescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	0ms	8035	2585
QuickSort Primeiro	0ms	9019	2656
QuickSort Último	0ms	8672	2640
QuickSort Aleatório	1ms	7680	2586
QuickSort Mediana	0ms	5886	2706

3.2.3 - Ordenação Completa Decrescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	0ms	7305	2583
QuickSort Primeiro	0ms	9531	2605
QuickSort Último	0ms	8943	2667
QuickSort Aleatório	0ms	8018	2609
QuickSort Mediana	0ms	6076	2653

3.2.4 - Ordenação Semi Completa

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	0ms	7597	2575
QuickSort Primeiro	0ms	8353	2644
QuickSort Último	0ms	9067	2652
QuickSort Aleatório	0ms	7785	2618
QuickSort Mediana	0ms	6515	2686

3.3 – Tabelas de Dados para Vetores de Tamanho 10.000

3.3.1 – Ordenação Aleatória

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	2ms	105531	33690
QuickSort Primeiro	1ms	118741	34339
QuickSort Último	1ms	126911	34306
QuickSort Aleatório	3ms	100195	34117
QuickSort Mediana	1ms	85294	34824

3.3.2 – Ordenação Completa Crescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	2ms	98491	33885
QuickSort Primeiro	1ms	147550	33546
QuickSort Último	1ms	129605	34063
QuickSort Aleatório	2ms	109156	33991
QuickSort Mediana	2ms	87555	34666

3.3.3 – Ordenação Completa Decrescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	1ms	96378	34059
QuickSort Primeiro	1ms	120110	34158
QuickSort Último	1ms	118987	34396
QuickSort Aleatório	3ms	111894	33972
QuickSort Mediana	2ms	85869	34701

3.3.4 – Ordenação Semi Completa

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	2ms	116947	33273
QuickSort Primeiro	1ms	125384	34133
QuickSort Último	2ms	128910	34006
QuickSort Aleatório	3ms	101044	34174
QuickSort Mediana	1ms	90697	34428

3.4 – Tabelas de Dados para Vetores de Tamanho 100.000

3.4.1 - Ordenação Aleatória

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	11ms	1329108	414293
QuickSort Primeiro	10ms	1580580	418720
QuickSort Último	10ms	1611870	417132
QuickSort Aleatório	28ms	1313737	419173
QuickSort Mediana	9ms	973780	419981

3.4.2 - Ordenação Completa Crescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	12ms	1430152	418382
QuickSort Primeiro	12ms	1462550	419117
QuickSort Último	12ms	1451808	415836
QuickSort Aleatório	25ms	1209203	414647
QuickSort Mediana	9ms	993671	418124

3.4.3 - Ordenação Completa Decrescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	11ms	1373856	419780
QuickSort Primeiro	11ms	1569303	417921
QuickSort Último	11ms	1582794	415870
QuickSort Aleatório	24ms	1268352	419071
QuickSort Mediana	10ms	988550	415874

3.4.4 - Ordenação Semi Completa

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	11ms	1362243	418602
QuickSort Primeiro	11ms	1512780	418329
QuickSort Último	11ms	1595565	415167
QuickSort Aleatório	23ms	1317321	416870
QuickSort Mediana	10ms	989040	416139

3.5 – Tabelas de Dados para Vetores de Tamanho 1.000.000

3.5.1 - Ordenação Aleatória

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	132ms	13119030	4123670
QuickSort Primeiro	136ms	15892474	4160811
QuickSort Último	133ms	16145483	4155091
QuickSort Aleatório	246ms	13395149	4120172
QuickSort Mediana	98ms	9916935	4178850

3.5.2 - Ordenação Completa Crescente

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	135ms	13215022	4158079
QuickSort Primeiro	138ms	15822243	4163934
QuickSort Último	138ms	15436328	4161892
QuickSort Aleatório	264ms	12159156	4143481
QuickSort Mediana	93ms	9901643	4167781

3.5.3 - Ordenação Completa Decrescente

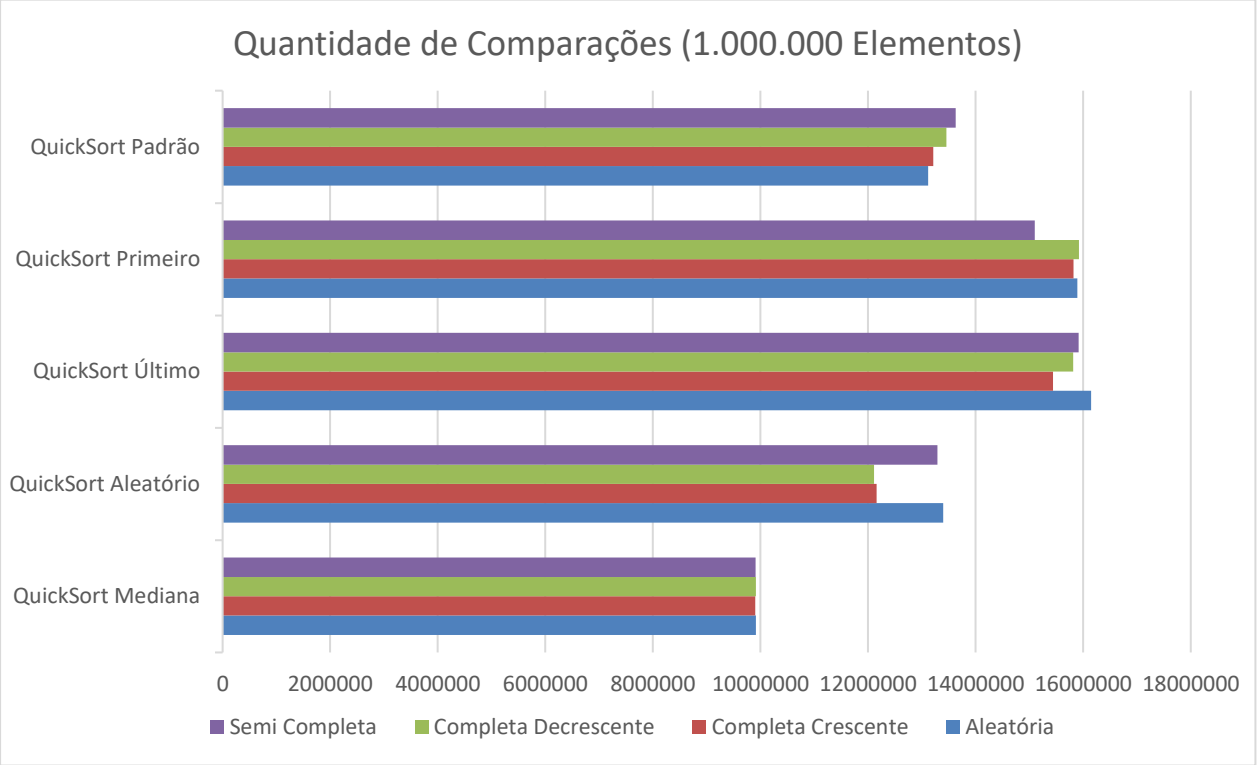
Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	134ms	13456029	4160310
QuickSort Primeiro	138ms	15918849	4158855
QuickSort Último	138ms	15815332	4168722
QuickSort Aleatório	265ms	12112138	4160025
QuickSort Mediana	101ms	9913308	4165423

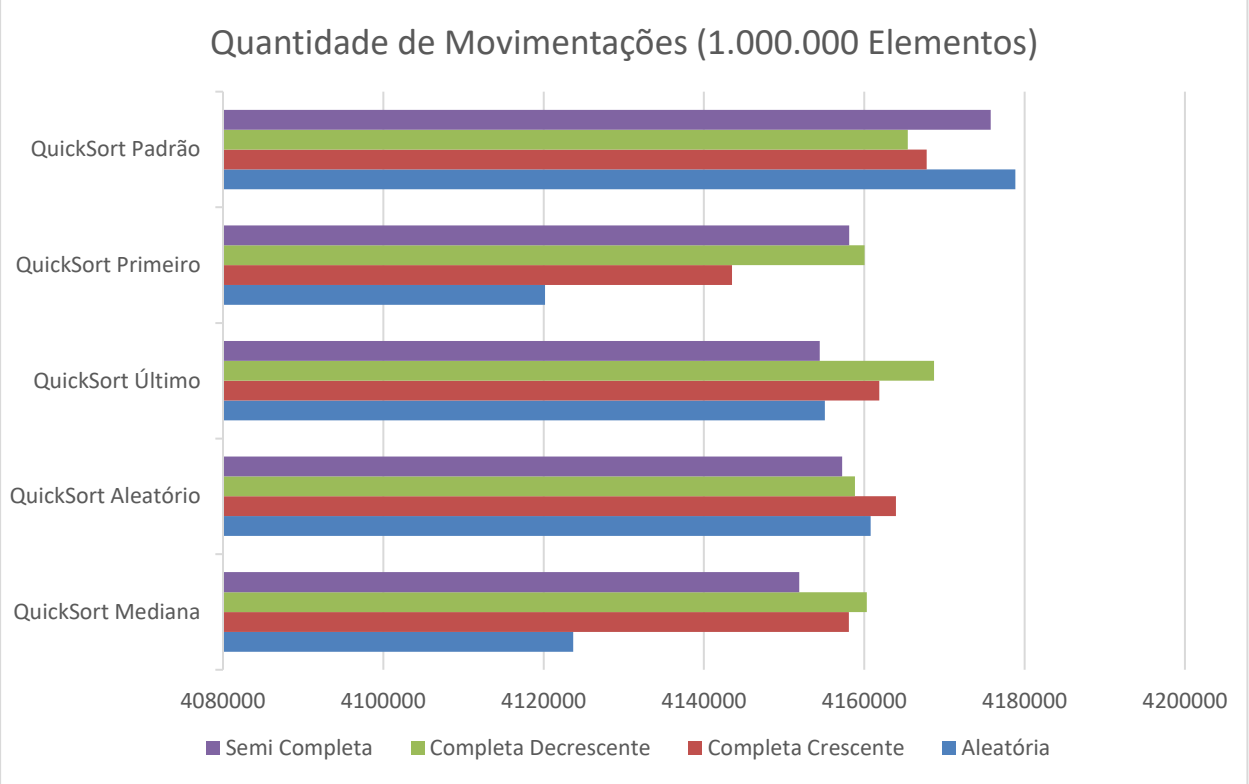
3.5.4 - Ordenação Semi Completa

Algoritmo	Execução	Comparações	Movimentações
QuickSort Padrão	135ms	13627587	4151901
QuickSort Primeiro	139ms	15102539	4157228
QuickSort Último	136ms	15914205	4154424
QuickSort Aleatório	258ms	13288056	4158112
QuickSort Mediana	101ms	9911162	4175769

4 – GRÁFICOS

A seguir vamos observar alguns desses dados em alguns gráficos de barra, para ilustrar melhor a diferença entre os cada abordagem, vamos analizar a quantidade de movimentações e comparações entre as diferentes propostas de pivô e para cada padrão de organização inicial do vetor. Abordaremos apenas o caso do vetor de tamanho 1.000.000, pois como temos presente muitos dados, os valores obtidos com esse tamnho de vetor se mostraram os mis interessantes.





5 – CONCLUSÃO

Com base nos dados de saída obtidos para as diferentes implementações de QuickSort e seus pivôs, podemos tirar algumas conclusões relevantes em termos de desempenho, comparações e movimentações. Vamos analisar cada uma das variações testadas e seu comportamento em diferentes cenários:

5.1 - Comparação Geral Entre os Pivôs:

- **Pivô mediana de três**: De maneira geral, a estratégia de mediana de três (comparando o primeiro, o meio e o último elemento) parece ser a mais eficiente em termos de comparações e movimentações na maioria dos cenários, principalmente para vetor grandes. Isso faz sentido, pois essa abordagem evita os piores cenários de particionamento desbalanceado.

- **Pivô aleatório**: Embora o pivô aleatório tenda a ter um desempenho aceitável em muitos casos, ele apresenta variações e pode ter tempos de execução mais altos em alguns cenários, como no vetor reverso de tamanho 1.000.000, onde o tempo foi significativamente maior.

- **Pivô padrão (central)**: O desempenho do pivô central é razoável, mas ele perde para a mediana de três e, em alguns casos, para o pivô aleatório. O número de comparações e movimentações é geralmente maior, especialmente em vetores ordenados e reversos.

- **Primeiro e último pivôs**: Escolher o primeiro ou o último elemento como pivô geralmente leva a um desempenho pior, principalmente para vetores ordenados ou quase ordenados. Nesses casos, o particionamento acaba sendo altamente desbalanceado, resultando em mais comparações e movimentações.

5.2 - Análise por Ordenação do Vetor:

- **Vetores aleatórios (Random)**:
  - Nos vetores aleatórios, o **pivô mediana** e o **pivô aleatório** tendem a ser os mais eficientes. Para vetores grandes (100.000 e 1.000.000), o pivô mediana obteve consistentemente os menores tempos de execução e o menor número de comparações.
  - O **pivô padrão**, o **pivô primeiro** e o **pivô último** têm um desempenho pior, principalmente em vetores maiores.

- **Vetores Ordenados (Sorted)**:
  - Nos vetores já ordenados, o **pivô mediana** novamente se destaca com o menor número de comparações e movimentações. O **pivô primeiro** e o **pivô último** performam mal, especialmente em vetores de tamanho 100.000 e 1.000.000, com tempos de execução relativamente maiores devido ao particionamento altamente desbalanceado.

- **Vetores Inversamente Ordenados (Reversed Sorted)**:
  - Assim como nos vetores ordenados, os pivôs de **primeiro** e **último** elementos resultam em um particionamento desbalanceado, causando um aumento no número de comparações e movimentações.
  - O **pivô mediana** ainda mantém o desempenho mais eficiente.

- **Vetores Quase Ordenados (Almost Sorted)**:
  - Os resultados são similares aos vetores ordenados. O **pivô mediana** continua se destacando com o menor número de comparações, enquanto o **pivô primeiro** e o **pivô último** performam mal.

5.3 - Comportamento por Tamanho do Vetor:

- Para vetores pequenos (100 e 1.000 elementos), o impacto da escolha do pivô não é tão perceptível em termos de tempo de execução, com todos os algoritmos sendo executados em tempos muito curtos (geralmente 0 ou 1ms).

- À medida que os tamanhos dos vetores aumentam (10.000, 100.000 e 1.000.000), a diferença de desempenho entre as estratégias de pivô fica mais clara. **Pivôs como o primeiro ou o último** têm um desempenho visivelmente pior, enquanto **o pivô mediana** se destaca como a estratégia mais eficiente, tanto em termos de comparações quanto de movimentações.

5.4 - Movimentações e Comparações:

- O número de **comparações e movimentações** tende a ser um bom indicador de eficiência. Em quase todos os casos, o **pivô mediana** apresenta a menor quantidade de comparações e movimentações, o que corrobora a eficiência dessa estratégia.

- O **pivô primeiro** e o **pivô último** frequentemente resultam em mais comparações e movimentações, devido ao particionamento desbalanceado, principalmente em vetores que já estão ordenados ou quase ordenados.

5.5 - Conclusão geral:

- A **estratégia de mediana de três** é a mais eficiente na maioria dos cenários, independentemente do tipo de array (random, sorted, reversed sorted ou almost sorted). Ela tem consistentemente o menor número de comparações, movimentações e tempos de execução, especialmente em vetores grandes.
- O **pivô aleatório** também tem um bom desempenho, mas pode ser imprevisível em alguns cenários.
- As estratégias que usam o **primeiro** ou o **último elemento como pivô** são as menos eficientes, particularmente para vetores ordenados ou quase ordenados. Elas levam a um número muito maior de comparações e movimentações e tempos de execução mais altos em vetores grandes.

Portanto, **a escolha do pivô afeta diretamente a eficiência do QuickSort**, e a escolha de um pivô balanceado (como a mediana de três) é crucial para garantir um desempenho consistente e eficiente, especialmente para vetores grandes e ordenados.

6 – CÓDIGO DE TESTE

```java
    // Função de troca de elementos
    private void swap(int[] array, int i, int j) {
        movimentacoes++;
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    // Função para medir o tempo de execução
    public void testSorting(int[] array, String strategy) {
        comparacoes = 0;
        movimentacoes = 0;
        long startTime = System.currentTimeMillis();
        switch (strategy) {
            case "standard": quicksortStandard(array, 0, array.length - 1); break;
            case "first": quicksortFirstPivot(array, 0, array.length - 1); break;
            case "last": quicksortLastPivot(array, 0, array.length - 1); break;
            case "random": quicksortRandomPivot(array, 0, array.length - 1); break;
            case "median": quicksortMedianOfThree(array, 0, array.length - 1); break;
        }
        long endTime = System.currentTimeMillis();
        System.out.println(strategy + " - Tempo de execução: " + (endTime - startTime) +
            "ms, Comparações: " + comparacoes + ", movimentações: " + movimentacoes);
    }
    // Função para gerar arrays aleatórios, ordenados, etc.
    public static int[] generateArray(int size, String type) {
        int[] array = new int[size];
        Random rand = new Random();
        for (int i = 0; i < size; i++) {
            array[i] = rand.nextInt(1000000);
        }
        switch (type) {
            case "sorted": java.util.Arrays.sort(array); break;
            case "reversed": java.util.Arrays.sort(array); reverse(array); break;
            case "almost_sorted": for (int i = 0; i < size / 10; i++)
                array[rand.nextInt(size)] = rand.nextInt(1000000); break;
        }
        return array;
    }
    public static void reverse(int[] array) {
        for (int i = 0; i < array.length / 2; i++) {
            int temp = array[i];
            array[i] = array[array.length - 1 - i];
            array[array.length - 1 - i] = temp;
        }
    }
    public static void main(String[] args) {
        QuickSortPivot quickSort = new QuickSortPivot();
        int[] sizes = {100, 1000, 10000, 100000, 1000000};
        String[] strategies = {"standard", "first", "last", "random", "median"};
        String[] types = {"Random", "Sorted", "Reversed Sorted", "Almost Sorted"};
        for (int size : sizes) {
            for (String type : types) {
                int[] array = generateArray(size, type);
                System.out.println("\nArray de tamanho " + size + " (" + type + ")");
                for (String strategy : strategies) {
                    int[] arrayCopy = array.clone(); // clonar o array para cada teste
                    quickSort.testSorting(arrayCopy, strategy);
                }
            }
        }
    }
}
```