Pontifícia Universidade Católica de Minas Gerais
Ciência da Computação
Algoritimo e Estrutura de Dados 2
Mariana Almeida Mendonça

1 Introdução

O Quicksort é um dos algoritmos de ordenação mais eficientes e amplamente utilizados em ciência da computação. Ele se baseia no paradigma de divisão e conquista, onde o array a ser ordenado é dividido em subarrays menores, e esses subarrays são ordenados recursivamente.

Este relatório analisa o desempenho do algoritmo Quicksort em diferentes cenários (arrays aleatórios, ordenados e quase ordenados) e com diferentes estratégias de escolha de pivô, comparando o tempo de execução de cada abordagem para arrays de tamanhos variados.

Para acesso ao código criado acesse: https://github.com/marialmeida1/study-aeds2/tree/main/labs/lab$_0 6$

2 Cenários e Estratégias

2.1 Cenários

Nesta análise, o algoritmo Quicksort foi testado em três cenários distintos de entrada de dados, cada um apresentando características específicas que afetam o desempenho do algoritmo. Os cenários considerados são:

- **Array Aleatório (Random):** Os elementos do array são dispostos de forma completamente aleatória, sem nenhuma ordenação prévia. O código realizado em *Java* utiliza da biblioteca *""java.util.Random""* para realizar a criação de um array randomico.
- **Array Ordenado (Sorted):** Todos os elementos do array já estão dispostos em ordem crescente. O código realizado em *Java* cria um array de acordo com a posição do array.

- **Array Quase Ordenado (Nearly Sorted):** Este cenário apresenta um array onde a maioria dos elementos já está em ordem crescente, mas com algumas trocas ou inversões mínimas. No código realizado em *Java* é criado um array ordenado e posteriormente é realizado algumas trocas dentro do array.

2.2 Estratégias

Para cada um dos cenários, o Quicksort foi executado utilizando quatro diferentes estratégias de escolha de pivô, descritas a seguir:

- **Primeiro elemento:** O pivô é escolhido como o primeiro elemento do array ou subarray.
- **Segundo elemento:** O pivô é o último elemento do array ou subarray.
- **Mediana de três elementos:** O pivô é escolhido como o elemento central (mediana) do array ou subarray. No código *Java* é realizado a mediana do primeiro elemento, do último elemento e do elemento do meio do array ou subarray, possibilitando criar uma mediana de custo baixo.
- **Elemento aleatório:** O pivô é selecionado de forma aleatória. No código realizado em *Java* a escolha é feita por meio da biblioteca *""java.util.Random""* procurando um elemento randomico.

3 Código

A implementação do código do algoritmo Quicksort foi estruturada de forma modular para permitir a escolha de diferentes estratégias de pivô. O algoritmo principal do método de ordenação é recursivo e segue o paradigma de divisão e conquista. O código no total apresenta os seguinte organização e custo:

1. **Construtor Quicksort:** $O(1)$
2. **Swap:** $O(1)$

3. **TypeQuickSort:** $O(1)$
4. **QuicksortAlgorithm:** $O(n * log(n)$
5. **QuicksortFirstPivot:** $O(1)$
6. **QuicksortLastPivot:** $O(1)$
7. **QuicksortRandom:** $O(1)$
8. **QuicksortMedianOfThree:** $O(1)$
9. **generateRandomArray:** $O(n)$
10. **generateOrderedArray:** $O(n)$
11. **generateNearlySortedArray:** $O(n)$
12. **runTests:** $O(O(n * log(n))$ *depende do método principal Quicksort*
13. **main:** $O(O(n * log(n))$ *depende do método principal Quicksort*

Conclui-se, portanto, que a análise de complexidade do código principal é $O(O(n * log(n))$, uma vez que o maior nível de complexidade identificado é $O(O(n * log(n))$. É importante ressaltar que os métodos mencionados nos *tópicos 9 a 12* são auxiliares e destinados à realização de testes, enquanto os demais métodos discutidos são fundamentais para a implementação do algoritmo principal.

4 Resultados

Os testes realizado avaliam o desempenho do algoritmo Quicksort em diferentes cenários de entrada, variando o tamanho do array e o tipo de ordenação. A tabela mostra os tempos de execução em nanosegundos (ns) para quatro opções diferentes de escolha de pivô, incluindo o primeiro, o último, um pivô aleatório e a mediana de três elementos.

Tipo de Array	Primeiro (ns)	Último (ns)	Random (ns)	Mediana (ns)
Random - 10	18551	28027	78052	30410
Sorted - 10	17796	17052	45419	20441
Nearly Sorted - 10	17975	11339	35240	14314
Random - 100	190443	476533	331297	135331
Sorted - 100	324053	217475	1014768	271802
Nearly Sorted - 100	23157	233075	346468	108226
Random - 1000	343409	5349612	1874107	635437
Sorted - 1000	4761633	2965472	339319	176368
Nearly Sorted - 1000	335855	2937102	356256	147017
Random - 10000	2820299	85652312	3007063	1292166
Sorted - 10000	77366572	83593768	1712705	543862
Nearly Sorted - 10000	1114069	104321862	1407534	501280

Table 1: Tempos de execução em nanosegundos para diferentes cenários.

4.1 Gráficos

4.1.1 Gráfico de todos dados

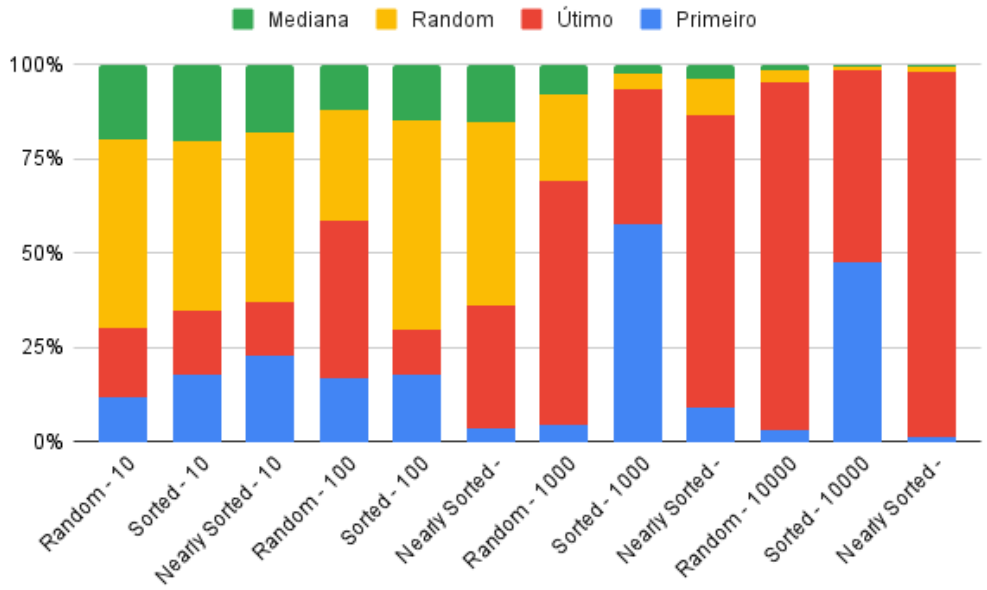

Figure 1: Dados de todos os resultados

4.1.2 Gráfico devido ao tamanho do array

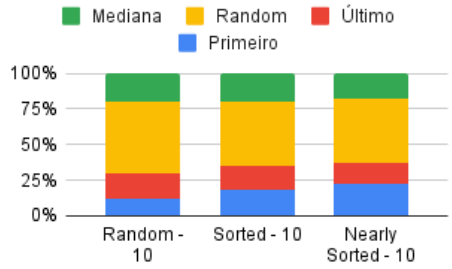

Figure 2: Array com 10 (dez) elementos

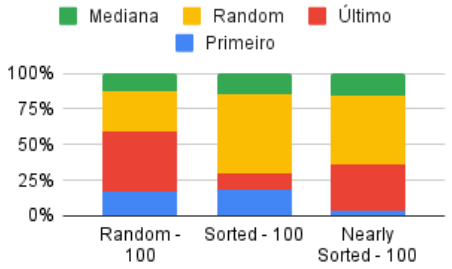

Figure 3: Array com 100 (cem) elementos

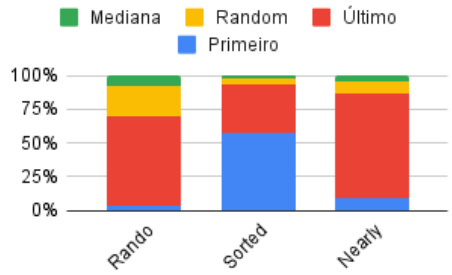

Figure 4: Array com 1000 (mil) elementos

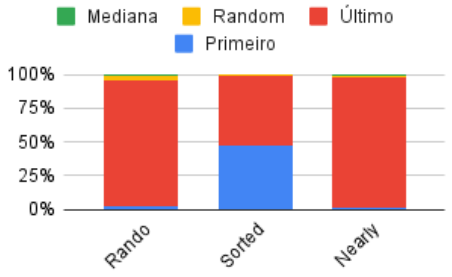

Figure 5: Array com 10000 (dez mil) elementos

4.2 Observações Gerais:

- **Tamanho do Array:** Os testes foram realizados para tamanhos de array de 10, 100, 1000 e 10000 elementos, permitindo observar como o algoritmo se comporta à medida que o volume de dados aumenta.
- **Tipos de Array:**
  - **Random:** Arrays com elementos gerados aleatoriamente.

  - **Sorted:** Arrays que já estão ordenados.
  - **Nearly Sorted:** Arrays que estão quase ordenados, com algumas trocas realizadas aleatoriamente.

- **Desempenho por Tipo de Array:**
  - **Arrays Aleatórios:** O tempo de execução tende a ser mais alto para tamanhos maiores, com a opção *último pivô* mostrando o pior desempenho, especialmente para o tamanho de 10.000 elementos.
  - **Arrays Ordenados:** O desempenho é geralmente mais eficiente em comparação com arrays aleatórios. A escolha do pivô afeta menos o tempo de execução, com a *último pivô* ainda apresentando um tempo considerável para arrays de 10.000 elementos.
  - **Arrays Quase Ordenados:** Os tempos de execução são variáveis, mas, em muitos casos, a *primeiro pivô* apresenta o melhor desempenho para entradas menores. Essa tendência indica que o Quicksort se beneficia de entradas que já estão parcialmente ordenadas.
- **Comparação entre Opções:**
  - A análise das diferentes opções de seleção de pivô revela que a escolha do pivô tem um impacto significativo no desempenho. Em geral, a *mediana de três* tendem a apresentar resultados mais favoráveis em arrays aleatórios e quase ordenados.
  - No entanto, a *último pivô* frequentemente resulta em tempos de execução mais altos, especialmente para arrays grandes e aleatórios. Vale ressaltar que arrays de grande tamanhos o *primeiro pivô* também apresenta dificuldades.