UNIVERSIDADE ESTADUAL DE SANTA CRUZ

ANÁLISE COMPARATIVA DE MÉTODOS DE ORDENAÇÃO DE VETORES

Antonio Henrique Oliveira Santos

Ariel Pina Ribeiro

Gabriel Rosa Galdino

Ilhéus - Bahia

2024

Sumário

Introdução

Neste relatório, analisamos o desempenho de quatro algoritmos de ordenação — Quick Sort, Bubble Sort, Shell Sort e Heap Sort — em três diferentes tamanhos de vetores (100, 1000 e 10000 elementos). O tempo de execução foi medido para diferentes números de repetições, a fim de obter uma média mais precisa dos tempos de execução. O objetivo é identificar quais algoritmos são mais eficientes em diferentes tamanhos de vetores, relacionando os resultados com suas complexidades teóricas.

Descrição dos Métodos

1. Quick Sort

   O Quick Sort é um algoritmo de ordenação eficiente baseado no paradigma de divisão e conquista. Basicamente a operação do algoritmo pode ser resumida na seguinte estratégia: divide sua lista de entrada em duas sub-listas a partir de um pivô, para em seguida realizar o mesmo procedimento nas duas listas menores até uma lista unitária.

   Complexidade:

   - Melhor caso: O(n log(n)).
   - Pior caso: $O(n^2)$ (Ocorre quando o pivô é mal escolhido).
   - Caso médio: O(n log(n)).

   Além disso, é um algoritmo eficiente, indicado para grandes conjuntos de dados devido à sua complexidade média(n log n).

2. Bubble Sort

   O Bubble Sort é um algoritmo de ordenação simples em que os pares consecutivos são comparados e trocados se estiverem na ordem errada. Esse processo é repetido até que o vetor esteja ordenado.

   Complexidade:

   - Melhor caso: O(n) (Ocorre quando o array já está ordenado).
   - Pior caso: $O(n^2)$ (Ocorre quando o vetor está em ordem inversa).
   - Caso médio: $O(n^2)$ (O vetor está em uma ordem aleatória).

   Além disso, é um algoritmo de método simples de ser implementado, porém ineficiente para grandes conjuntos de dados, funciona melhor para conjuntos pequenos.

3. Shell Sort

   O Shell Sort é um algoritmo que funciona como uma extensão do Insertion Sort, que compara e troca elementos em posições distantes (gap), reduzindo progressivamente o gap até 1. Diferente do insertion que possui a troca de itens adjacentes para determinar o ponto de inserção.

   Complexidade:

   - Melhor caso: $O(n \log^2(n))$. (Dependendo do esquema de gaps).
   - Pior caso: Varia de $O(n^{3/2})$ a $O(n^2)$.

   Além disso, é um algoritmo bom para ordenar um número moderado de elementos e quando encontra um arquivo parcialmente ordenado

trabalha menos, mas geralmente é inferior ao Quick Sort em grandes conjuntos.

4. Heap Sort

   O Heap Sort é um algoritmo que se baseia na estrutura de dados conhecida como heap. Um heap é uma árvore binária especial em que cada nó possui um valor maior ou igual ao valor de seus filhos (se for um heap máximo) ou menor ou igual (se for um heap mínimo). Ele constrói um heap máximo (ou mínimo), onde o maior (ou menor) elemento é removido e colocado no final (ou início) do array repetidamente.

   Complexidade:

   - Melhor, pior e caso médio: O(n log(n)).

   Além disso, é um algoritmo que possui um método de ordenação consistente e eficiente, no qual aplicações que não podem tolerar eventuais variações no tempo esperado de execução devem usar o Heap Sort, porém possui uma sobrecarga maior em relação ao Quick Sort pois para garantir as propriedades do heap, as operações de construção do heap (heapify) e reestruturação após cada remoção exigem múltiplas comparações e trocas. Embora essas operações sejam O(n log(n)), a constante associada pode ser maior devido à complexidade dos acessos e trocas dentro da árvore.

Metodologia

***Configuração do equipamento utilizado nos testes**

- Modelo do Notebook: Lenovo IdeaPad Gaming 3i.
- Processador: Intel Core i5-11300H (4 núcleos, 8 threads, frequência base de 3,10 GHz, frequência máxima de 4,40 GHz).
- Sistema Operacional: Ubuntu.

Tamanho dos Vetores

Para a realização da análise dos algoritmos de ordenação foram testados 3 tamanhos diferentes de vetores:

- **100 elementos:** Representa um pequeno conjunto de dados.
- **1.000 elementos:** Considerado um conjunto médio de dados.

- **10.000 elementos:** Representa um grande conjunto de dados para avaliar escalabilidade.

Os valores dos vetores foram previamente carregados de um arquivo chamado `seed_01.dat`. Este arquivo contém 10.000 números inteiros desordenados no intervalo de 0 a 100.000, garantindo consistência nos dados entre as execuções.

Medição do Tempo

Para medir o tempo de execução do algoritmo foi utilizada a função **`clock()`** da biblioteca padrão **time.h**.

A medição foi realizada da seguinte forma:

```
start = clock();
for (int j = 0; j < repetitions; j++) {
    quickSort(tempArray, 0, size - 1);
}
end = clock();
printf("Quick Sort: %.6f segundos\n", (double)(end - start) /
CLOCKS_PER_SEC / repetitions);
free(tempArray);
```

- Antes da execução do algoritmo, o tempo inicial foi registrado com o **start**.
- Após a execução o tempo final é registrado em **end**.
- O cálculo do tempo de execução é dado por:

  Tempo = $\frac{(end - start)}{CLOCKS\_PER\_SEC * repetitions}$

  Essa abordagem mede o tempo médio de execução ao longo de várias repetições.

Número de Execuções

Os algoritmos foram testados com 1, 10 e 100 execuções para cada tamanho de vetor, com o objetivo de analisar o impacto do overhead inicial e medir a precisão em diferentes cenários. Além disso, o código foi compilado e executado pelo menos duas vezes para cada configuração, a fim de observar possíveis otimizações automáticas do compilador e avaliar como o desempenho muda entre a primeira e a segunda execução.

Execução do Programa

Para cada algoritmo e tamanho de vetor:

1. O vetor original foi carregado do arquivo e duplicado em um array temporário.
2. O algoritmo foi executado sobre o array temporário.
3. Após cada execução, o array temporário foi descartado e recriado a partir do vetor original.

Essa abordagem garante que cada algoritmo opera sobre dados consistentes e sem interferência de execuções anteriores.

Resultados

1° Interação de resultados:

1 Repetição

$\frac{Tamanho}{Algoritmo}$	100	1000	10000
Quick Sort	0.000007s	0.000074s	0.000967s
Bubble Sort	0.000019s	0.001466s	0.150360s
Shell Sort	0.000007s	0.000115s	0.001645s
Heap Sort	0.000009s	0.000114s	0.001535s

10 Repetições

$\frac{Tamanho}{Algoritmo}$	100	1000	10000
Quick Sort	0.000002s	0.000037s	0.000415s
Bubble Sort	0.000011s	0.000934s	0.092223s
Shell Sort	0.000002s	0.000030s	0.000432s
Heap Sort	0.000013s	0.000105s	0.001180s

100 Repetições

$\frac{Tamanho}{Algoritmo}$	100	1000	10000
Quick Sort	0.000001s	0.000022s	0.000315s
Bubble Sort	0.000010s	0.000891s	0.088124s
Shell Sort	0.000001s	0.000022s	0.000314s
Heap Sort	0.000005s	0.000085s	0.001127s

2° Interação de resultados:

1 Repetição

$\frac{Tamanho}{Algoritmo}$	100	1000	10000
Quick Sort	0.000006s	0.000071s	0.000949s
Bubble Sort	0.000018s	0.001414s	0.139839s
Shell Sort	0.000007s	0.000106s	0.001527s
Heap Sort	0.000008s	0.000104s	0.001411s

10 Repetições

$\frac{Tamanho}{Algoritmo}$	100	1000	10000
Quick Sort	0.000003s	0.000052s	0.000483s
Bubble Sort	0.000011s	0.000961s	0.091035s
Shell Sort	0.000002s	0.000030s	0.000426s
Heap Sort	0.000007s	0.000093s	0.001180s

100 Repetições

$\frac{Tamanho}{Algoritmo}$	100	1000	10000
Quick Sort	0.000002s	0.000022s	0.000329s
Bubble Sort	0.000011s	0.000900s	0.088062s
Shell Sort	0.000001s	0.000022s	0.000308s
Heap Sort	0.000005s	0.000080s	0.001114s

Gráficos

Como estava sendo trabalhado com valores pequenos, convertemos os valores de segundos para microssegundos (μs) e depois foi feita a média das repetições e seus resultados convertidos em escala logarítmica.

Cálculos Abaixo:

1° Interação

Com 1 repetição

Tamanho do Array: 100

- Quick Sort: 7 µs
- Bubble Sort: 19 µs
- Shell Sort: 7 µs
- Heap Sort: 9 µs

Tamanho do Array: 1000

- Quick Sort: 74 µs
- Bubble Sort: 1466 µs
- Shell Sort: 115 µs
- Heap Sort: 114 µs

Tamanho do Array: 10000

- Quick Sort: 967 µs
- Bubble Sort: 150360 µs
- Shell Sort: 1645 µs
- Heap Sort: 1535 µs

Com 10 repetições

Tamanho do Array: 100

- Quick Sort: 2 µs
- Bubble Sort: 11 µs
- Shell Sort: 2 µs
- Heap Sort: 13 µs

Tamanho do Array: 1000

- Quick Sort: 37 µs
- Bubble Sort: 934 µs
- Shell Sort: 30 µs
- Heap Sort: 105 µs

Tamanho do Array: 10000

- Quick Sort: 415 µs
- Bubble Sort: 92223 µs
- Shell Sort: 432 µs
- Heap Sort: 1180 µs

Com 100 repetições

Tamanho do Array: 100

- Quick Sort: 1 µs
- Bubble Sort: 10 µs
- Shell Sort: 1 µs
- Heap Sort: 5 µs

Tamanho do Array: 1000

- Quick Sort: 22 µs
- Bubble Sort: 891 µs
- Shell Sort: 22 µs
- Heap Sort: 85 µs

Tamanho do Array: 10000

- Quick Sort: 315 µs
- Bubble Sort: 88124 µs
- Shell Sort: 314 µs
- Heap Sort: 1127 µs

Média Final (em microssegundos):

- ➢ **Quick Sort:**
  - Para 100: (7 + 2 + 1) / 3 ≈ 3.33 µs
  - Para 1000: (74 + 37 + 22) / 3 ≈ 44.33 µs
  - Para 10000: (967 + 415 + 315) / 3 ≈ 565.67 µs
- ➢ **Bubble Sort:**
  - Para 100: (19 + 11 + 10) / 3 ≈ 13.33 µs
  - Para 1000: (1466 + 934 + 891) / 3 ≈ 1097.00 µs
  - Para 10000: (150360 + 92223 + 88124) / 3 ≈ 110902.33 µs
- ➢ **Shell Sort:**
  - Para 100: (7 + 2 + 1) / 3 ≈ 3.33 µs
  - Para 1000: (115 + 30 + 22) / 3 ≈ 55.67 µs
  - Para 10000: (1645 + 432 + 314) / 3 ≈ 797.00 µs
- ➢ **Heap Sort:**
  - Para 100: (9 + 13 + 5) / 3 ≈ 9.00 µs

- Para 1000: (114 + 105 + 85) / 3 ≈ 101.33 µs
- Para 10000: (1535 + 1180 + 1127) / 3 ≈ 1280.67 µs

Cálculo Logarítmico (Log_{10}):

Para cada valor, calculemos o logaritmo na base 10:

Quick Sort:

- Para 100: log(3.33) ≈ 0.522
- Para 1000: log(44.33) ≈ 1.647
- Para 10000: log(565.67) ≈ 2.752

Bubble Sort:

- Para 100: log(13.33) ≈ 1.125
- Para 1000: log(1097.00) ≈ 3.041
- Para 10000: log(110902.33) ≈ 5.045

Shell Sort:

- Para 100: log(3.33) ≈ 0.522
- Para 1000: log(55.67) ≈ 1.745
- Para 10000: log(797.00) ≈ 2.902

Heap Sort:

- Para 100: log(9.00) ≈ 0.954
- Para 1000: log(101.33) ≈ 2.006
- Para 10000: log(1280.67) ≈ 3.107

2° Interação

1 repetição:

- ➢ **Quick Sort:**
  - Para 100: 6.5 µs
  - Para 1000: 71.0 µs
  - Para 10000: 949.0 µs
- ➢ **Bubble Sort:**
  - Para 100: 18.0 µs
  - Para 1000: 414.0 µs
  - Para 10000: 139839.0 µs
- ➢ **Shell Sort:**
  - Para 100: 7.0 µs

- Para 1000: 106.0 µs
- Para 10000: 1527.0 µs

➢ **Heap Sort:**

- Para 100: 8.0 µs
- Para 1000: 104.0 µs
- Para 10000: 1411.0 µs

---

10 repetições:

➢ **Quick Sort:**

- Para 100: 3.0 µs
- Para 1000: 52.0 µs
- Para 10000: 483.0 µs

➢ **Bubble Sort:**

- Para 100: 11.0 µs
- Para 1000: 961.0 µs
- Para 10000: 91035.0 µs

➢ **Shell Sort:**

- Para 100: 2.0 µs
- Para 1000: 30.0 µs
- Para 10000: 426.0 µs

➢ **Heap Sort:**

- Para 100: 7.0 µs
- Para 1000: 93.0 µs
- Para 10000: 1180.0 µs

---

100 repetições:

➢ **Quick Sort:**

- Para 100: 2.0 µs
- Para 1000: 22.0 µs
- Para 10000: 329.0 µs

➢ **Bubble Sort:**

- Para 100: 11.0 µs
- Para 1000: 900.0 µs
- Para 10000: 88062.0 µs

➢ **Shell Sort:**

- Para 100: 1.0 µs
- Para 1000: 22.0 µs
- Para 10000: 308.0 µs

➢ **Heap Sort:**

- Para 100: 5.0 µs
- Para 1000: 80.0 µs
- Para 10000: 1114.0 µs

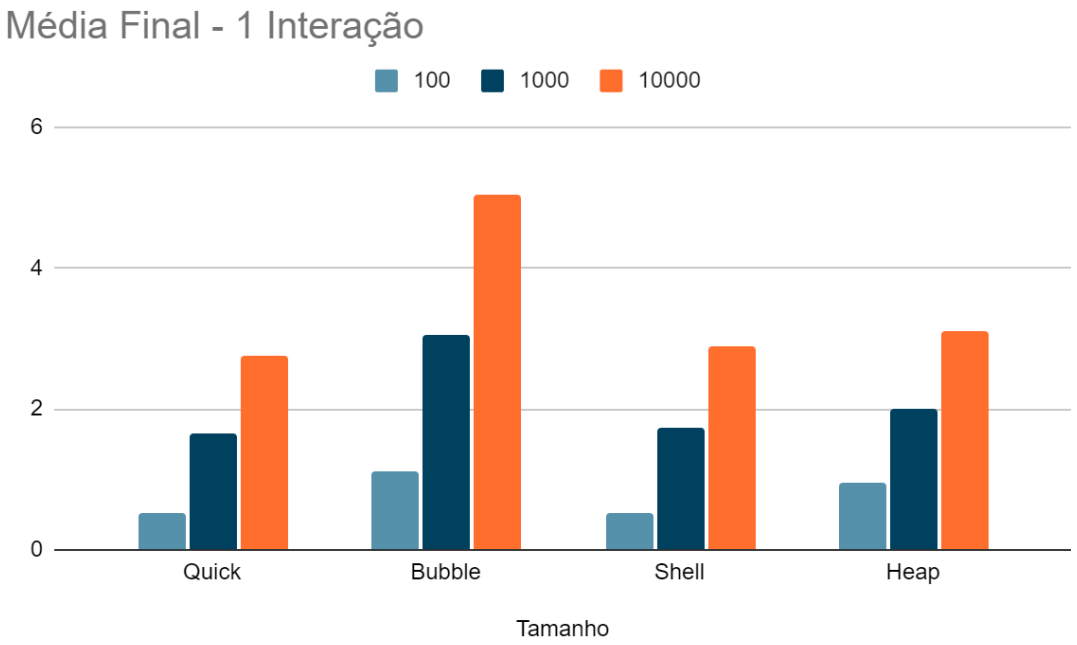
Média dos tempos (em microssegundos):

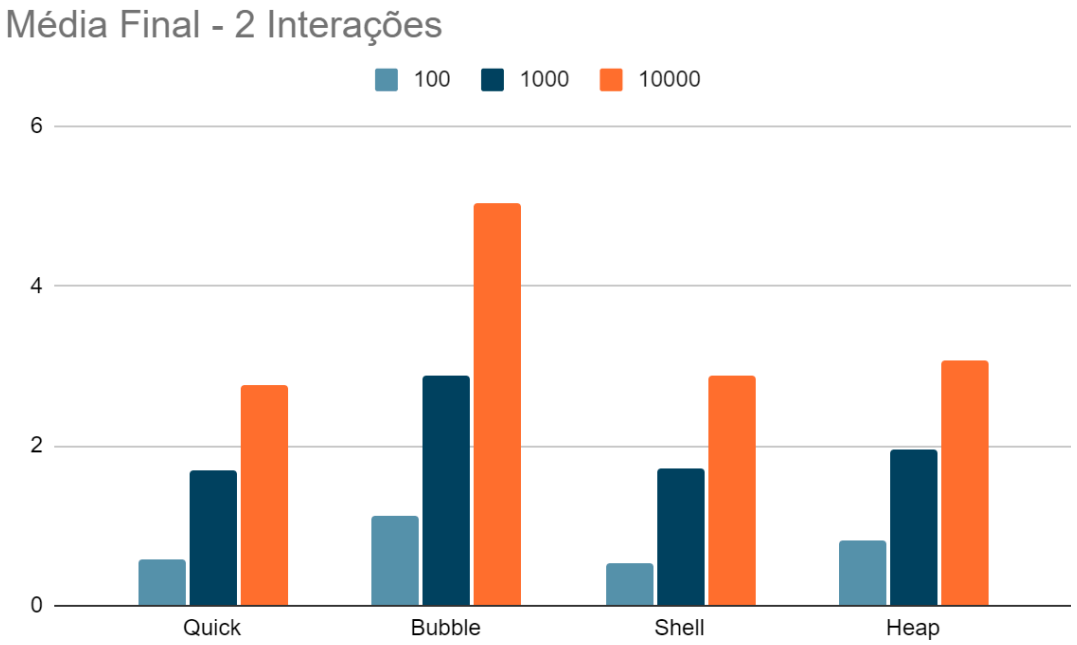
- **Quick Sort:**
  - Para 100: (6.5 + 3.0 + 2.0) / 3 ≈ 3.83 µs
  - Para 1000: (71.0 + 52.0 + 22.0) / 3 ≈ 48.33 µs
  - Para 10000: (949.0 + 483.0 + 329.0) / 3 ≈ 587.67 µs
- **Bubble Sort:**
  - Para 100: (18.0 + 11.0 + 11.0) / 3 ≈ 13.33 µs
  - Para 1000: (414.0 + 961.0 + 900.0) / 3 ≈ 758.33 µs
  - Para 10000: (139839.0 + 91035.0 + 88062.0) / 3 ≈ 106312.00 µs
- **Shell Sort:**
  - Para 100: (7.0 + 2.0 + 1.0) / 3 ≈ 3.33 µs
  - Para 1000: (106.0 + 30.0 + 22.0) / 3 ≈ 52.67 µs
  - Para 10000: (1527.0 + 426.0 + 308.0) / 3 ≈ 753.67 µs
- **Heap Sort:**
  - Para 100: (8.0 + 7.0 + 5.0) / 3 ≈ 6.67 µs
  - Para 1000: (104.0 + 93.0 + 80.0) / 3 ≈ 92.33 µs
  - Para 10000: (1411.0 + 1180.0 + 1114.0) / 3 ≈ 1201.67 µs

Agora, aplicamos o logaritmo na base 10 nas médias dos tempos:

- **Quick Sort:**
  - Para 100: log(3.83) ≈ 0.584
  - Para 1000: log(48.33) ≈ 1.684
  - Para 10000: log(587.67) ≈ 2.769
- **Bubble Sort:**
  - Para 100: log(13.33) ≈ 1.126
  - Para 1000: log(758.33) ≈ 2.879
  - Para 10000: log(106312.00) ≈ 5.026
- **Shell Sort:**
  - Para 100: log(3.33) ≈ 0.522
  - Para 1000: log(52.67) ≈ 1.721
  - Para 10000: log(753.67) ≈ 2.877
- **Heap Sort:**
  - Para 100: log(6.67) ≈ 0.824
  - Para 1000: log(92.33) ≈ 1.964
  - Para 10000: log(1201.67) ≈ 3.079

Resultados:





Discussão

Análise dos Resultado

Com base nos tempos de execução apresentados, os resultados mostram diferenças claras entre os métodos para diversos tamanhos de vetor.

1. **Vetores pequenos (n = 100):**
    - Para o tamanho de vetor 100, os tempos de execução são muito pequenos e bastante próximos entre os algoritmos. O Quick Sort e o Shell Sort apresentam os melhores desempenhos (0.000007 segundos) e são os mais rápidos.
    - O Bubble Sort tem um desempenho um pouco inferior (0.000019 segundos), mas ainda assim está em uma faixa muito pequena de tempo. A diferença entre os métodos é insignificante neste tamanho de vetor, visto que todos os tempos estão na ordem de microssegundos.
2. **Vetores médios (n = 1000):**
    - O Quick Sort continua se destacando, com 0.000074 segundos em 1 repetição. A diferença se torna mais notável à medida que o vetor cresce.
    - O Bubble Sort continua sendo significativamente mais lento, confirmando sua ineficiência para tamanhos maiores de vetores.
3. **Vetores grandes (n = 10000):**
    - O Quick Sort mantém sua eficiência, sendo o método mais rápido em todas as interações.
    - O Shell Sort também é eficiente, rivalizando com o Quick Sort em alguns casos.
    - O Heap Sort é competitivo, mas apresenta tempos ligeiramente superiores ao Shell Sort.
    - O Bubble Sort é claramente o mais ineficiente, levando cerca de 100 vezes mais tempo que o Quick Sort para tamanhos grandes.

Eficiência para Vetores Pequenos

1. Quick Sort e Shell Sort são os mais eficientes, devido à menor sobrecarga computacional para tamanhos pequenos.
2. O Heap Sort é competitivo, mas não supera os dois anteriores.
3. O Bubble Sort, por ser $O(n^2)$, é significativamente mais lento mesmo para vetores pequenos.

Eficiência para Vetores Maiores

1. Quick Sort é consistentemente o mais eficiente, aproveitando sua complexidade média O(n log(n)) e a escolha inteligente de pivôs.
2. Shell Sort também é muito eficiente, particularmente devido ao bom desempenho para tamanhos médios e grandes de vetores.
3. Heap Sort é confiável, mas menos eficiente que o Quick Sort para grandes vetores, principalmente devido ao custo de construção e manutenção do heap.

4. O Bubble Sort é extremamente ineficiente para tamanhos maiores, demonstrando sua inadequação para este cenário.

Algoritmo	Complexidade Média	Vetores Pequenos	Vetores Médios	Vetores Grandes
Quick Sort	O(n log(n))	Muito rápido	Muito rápido	Excelente
Shell Sort	$O(n \log^2(n))$	Muito rápido	Rápido	Rápido
Heap Sort	O(n log(n))	Rápido	Rápido	Bom
Bubble Sort	$O(n^2)$	Lento	Muito lento	Ineficiente

Observação: os algoritmos com complexidade O(n) têm desempenho pior à medida que o tamanho do vetor cresce, enquanto algoritmos O(n log(n)) são escaláveis.

Conclusão

Com base nos resultados obtidos, foi possível avaliar a eficiência de diferentes algoritmos de ordenação em cenários variados. A análise demonstrou que o desempenho de cada método está diretamente relacionado à sua complexidade de tempo e ao tamanho do vetor de entrada, evidenciando a superioridade de algoritmos mais avançados, como o Quick Sort, em comparação a métodos mais simples, como o Bubble Sort. Essa conclusão foi fundamentada no desempenho geral dos algoritmos, no impacto do tamanho dos vetores e no número de repetições, destacando os seguintes pontos observados:

1. Desempenho Geral
    - O Quick sort é o mais rápido para todos os tamanhos aleatórios experimentados.
    - O Bubble Sort, devido à sua complexidade quadrática ($O(n^2)$) é inviável para uso em grandes conjuntos de dados.
    - O Shell Sort e o Heap Sort apresentaram desempenhos similares entre si, sendo opções intermediárias, mas inferiores ao Quick Sort.
2. Impacto do Tamanho do Vetor
    - Para vetores pequenos, as diferenças de desempenho entre os algoritmos foram mínimas, com tempos de execução na faixa de microssegundos.

- Para vetores médios e grandes, o impacto da complexidade de tempo de cada algoritmo se tornou evidente. O Bubble Sort apresentou tempos exponencialmente maiores, enquanto os outros algoritmos mantiveram tempos mais consistentes.

3. Impacto do Número de Repetições
    - O aumento no número de repetições reduziu ainda mais o tempo médio por execução, especialmente para o Quick Sort. No entanto, o Bubble Sort apresentou um crescimento expressivo no tempo total de execução, enquanto os demais algoritmos mantiveram um desempenho estável e consistente.

Em suma, os resultados confirmam a teoria sobre as complexidades de tempo dos algoritmos. O Quick Sort se destaca como a melhor escolha para vetores grandes, enquanto o Bubble Sort se mostra adequado apenas para conjuntos de dados pequenos. O Shell Sort e o Heap Sort oferecem um equilíbrio entre simplicidade e eficiência, mas ainda apresentam um desempenho inferior ao do Quick Sort, especialmente quando aplicados a grandes volumes de dados.

Referências

1. **GEKSFORGEEKS**. *Heap Sort*. Disponível em: https://www.geeksforgeeks.org/heap-sort/. Acesso em: 1 dez. 2024.
2. **CALAZANS**, Luiz. *Entendendo o Heapsort*. Medium, 2024. Disponível em: https://luizcalazans.medium.com/entendendo-o-heapsort-95ec851dcdbf. Acesso em: 1 dez. 2024.
3. **TREINAWEB**. *Conheça os principais algoritmos de ordenação*. Disponível em: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao#:~:text=O%20Algoritmo%20Quicksort%2C%20criado%20por,Provavelmente%20é%20o%20mais%20utilizado. Acesso em: 1 dez. 2024.
4. **OPENDSA SERVER**. *Shellsort*. Virginia Tech, 2024. Disponível em: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Shellsort.html. Acesso em: 1 dez. 2024.
5. **DEVFURIA**. *Exemplos na linguagem C do algoritmo Bubble Sort*. Disponível em: http://devfuria.com.br/logica-de-programacao/exemplos-na-linguagem-c-do-algoritmo-bubble-sort/. Acesso em: 1 dez. 2024.
6. **SCIELO**. *A complexidade do algoritmo ShellSort*. Disponível em: https://www.scielo.br/j/tema/a/G7Ybrdy6w4K6gFYRvSj4tft/?lang=pt#:~:text=A%20complexidade%20de%20pior%20caso%20do%20algoritmo%20

ShellSort%20para%20qualquer,n%202%20log%20log%20n%20)%20. Acesso em: 1 dez. 2024.