

HeapSort e CycleSort

Adriano Ocampos, Artur da Silva e João Victor Dreissig

Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense – Campus Gravataí

Adrianosilveira.gr034@academico.if sul.edu.br

Artursilva.gr021@academico.if sul.edu.br

Joaomelo.gr017@academico.if sul.edu.br

Keywords:

- 1- C;
- 2- Ordenação;
- 3- Estabilidade;
- 4- Ordenação In-Place;
- 5- Algoritmos Híbridos.

Introdução:

A ordenação, segundo cientistas da tecnologia, é um problema fundamental no estudo de algoritmos, dada a sua importância para otimizar operações de dados. Os algoritmos de ordenação operam predominantemente sobre estruturas de dados lineares, como vetores e listas, nas quais cada elemento (ou nodo) mantém uma explícita ou implícita relação de vizinhança com os demais. O processo é definido pela transformação de um conjunto de entrada desordenado no mesmo conjunto de saída, porém reestruturado para satisfazer uma relação de ordem específica.

A utilidade primária da ordenação reside no fato de que dados organizados permitem a execução de buscas e fusões de maneira drasticamente mais eficiente, por exemplo, a busca binária, sendo, portanto, uma etapa crucial para a otimização de sistemas de informação, bancos de dados e algoritmos de processamento de dados em geral.

Desenvolvimento:

O **HeapSort** é um algoritmo de ordenação por comparação que utiliza uma heap binária. Sua principal vantagem é a complexidade de tempo de pior caso, que é garantidamente $O(n \log n)$, oferecendo um desempenho consistente. Por ser in-place, ele não requer espaço de memória adicional significativo, sendo ideal para sistemas com recursos limitados. No entanto, é um algoritmo instável, não preservando a ordem relativa de elementos com chaves iguais.

O **CycleSort** é um algoritmo de ordenação que se destaca por ser teoricamente ótimo em termos de número de escritas (trocas) na memória. Ele opera identificando e girando os "ciclos" de elementos fora do lugar, garantindo que cada elemento seja movido para sua posição final correta no máximo uma vez. Essa característica o torna extremamente valioso em situações em que as operações de escrita são custosas ou limitadas. Embora sua complexidade de tempo geral seja tipicamente $O(n^2)$ devido à busca pela posição correta, para arrays com elementos

distintos em um intervalo fixo, ele pode ser adaptado para atingir $O(n)$, focando sempre na minimização das escritas.

Resultados:

HeapSort

O experimento realizado buscou avaliar a eficiência e a estabilidade do algoritmo de ordenação HeapSort em três diferentes tamanhos de vetor e sob três condições iniciais: vetor aleatório, vetor crescente (já ordenado) e vetor decrescente (ordenado inversamente). O resultado, consolidado na tabela abaixo, reflete com precisão a natureza do HeapSort como um algoritmo de ordenação de tempo consistente, caracterizado pela complexidade $O(n \log n)$.

Tamanho do Vetor (N)	Tipo de Vetor	Média (s)	Desvio Padrão (s)
1.000	Aleatório	0.00000	0.000000
1.000	Crescente	0.00000	0.000000
1.000	Decrescente	0.00000	0.000000
100.000	Aleatório	0.025800	0.005074
100.000	Crescente	0.018550	0.008192
100.000	Decrescente	0.018550	0.003967
1.000.000	Aleatório	0.317150	0.006722
1.000.000	Crescente	0.227550	0.013489
1.000.000	Decrescente	0.235000	0.030505

AMBIENTE DE TESTE

Processador – Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz (3.60 GHz)

Memória RAM – 16,0 GB

Armazenamento – SSD SATA 480GB

O desempenho do HeapSort, conforme os dados apresentados, é altamente consistente e confirma plenamente sua complexidade teórica $O(n \log n)$. Para o vetor de 1.000 elementos, o tempo é insignificante (0.00000 s), tornando-se informativo a partir de 100.000 e 1.000.000 elementos. Nesses tamanhos maiores, o algoritmo demonstrou ser insensível à ordem inicial do vetor (aleatório, crescente ou decrescente), com os tempos de execução sempre na mesma ordem de magnitude (por exemplo, 0.23 s a 0.317 s para $N=1.000.000$).

A prova de sua eficiência reside na escalabilidade: ao aumentar o volume de dados em 10 vezes (de 100.000 para 1.000.000), o tempo de execução aumentou por um fator de aproximadamente 12 vezes. Este crescimento é uma confirmação quase exata do comportamento esperado de um algoritmo $O(n \log n)$, atestando a robustez e a alta eficiência do HeapSort para grandes volumes de dados.

CycleSort

A presente avaliação buscou determinar a performance e a consistência do algoritmo de ordenação CycleSort em diferentes conjuntos de dados: vetores de 1.000, 100.000 e 1.000.000 de elementos, nas configurações Crescente, Decrescente e Aleatório. Os resultados tabulados abaixo reforçam a classificação do CycleSort como um algoritmo de tempo de execução quadrático, ou $O(n^2)$.

Tamanho do Vetor (N)	Tipo de Vetor	Média (s)	Desvio Padrão (s)
1.000	Crescente	0.00000	0.00000
1.000	Decrescente	0.00075	0.00327
1.000	Aleatório	4.13480	0.00883
100.000	Crescente	4.13480	0.00883
100.000	Decrescente	7.51155	-
100.000	Aleatório	-	-
1.000.000	Crescente	-	-
1.000.000	Decrescente	-	-

1.000.000	Aleatório	-	-
-----------	-----------	---	---

A performance do CycleSort evidenciou ser fortemente influenciada tanto pelo volume de dados quanto pela organização inicial do vetor, o que é um traço marcante da complexidade $O(n^2)$. Embora tenha ordenado rapidamente os vetores crescente (0.00000 s) e decrescente (0.00075 s) de 1.000 elementos, o caso aleatório para o mesmo tamanho já demandou um tempo excessivo de 4.13480s. Esse problema de escalabilidade foi severo ao se mover para 100.000 elementos, onde o caso crescente consumiu 4.13480s e o decrescente, mesmo incompleto, atingiu 7.51155s. É notório que para 100.000 elementos aleatórios e para todos os testes com 1.000.000 de elementos, não houve conclusão da execução. Essa incapacidade de processar grandes conjuntos de dados confirma que o CycleSort não é uma escolha viável para aplicações de ordenação em larga escala, sendo rapidamente limitado pela sua complexidade de tempo quadrática.

Arquitetura/Implementação:

O sistema de ordenação foi desenvolvido através da implementação de dois algoritmos de comparação com funcionalidades distintas. O HeapSort foi escolhido como a solução principal para escalabilidade e desempenho consistente, baseando-se em uma heap binária e operando in-place. Sua arquitetura garante uma complexidade de tempo de $O(n \log n)$ em todos os casos, o que o torna insensível à ordem inicial dos dados e altamente eficiente para grandes volumes, conforme atestado pelos resultados (ex: tempos na casa de 0.23s a 0.317s para 1.000.000 de elementos). Em contraste, o CycleSort foi implementado visando um nicho específico: a otimização do número de escritas (trocas) na memória, onde cada elemento é movido no máximo uma vez para sua posição correta. Apesar de ser ótimo para minimizar I/O, sua complexidade de $O(n^2)$ mostrou-se um limitador severo na prática, com falhas na conclusão de testes para 100.000 elementos aleatórios e 1.000.000 de elementos em qualquer configuração, tornando-o inviável para ordenação em larga escala. A escolha entre os dois reflete o balanceamento entre a alta velocidade de processamento do HeapSort e a minimização de operações de escrita do CycleSort.

Conclusão:

O estudo abordou a ordenação como um problema algorítmico fundamental para otimização de dados, com foco na avaliação prática do HeapSort e do CycleSort. O HeapSort, implementado como uma solução in-place baseada em heap binária, destacou-se por sua alta escalabilidade e consistência, comprovando sua complexidade $O(n \log n)$. Os resultados

mostraram que o HeapSort é insensível à ordem inicial do vetor e altamente eficiente para 1.000.000 de elementos, apresentando um aumento de tempo de apenas cerca de 12 vezes ao aumentar o volume de dados em 10 vezes, o que o qualifica como a solução ideal para processamento de grandes volumes de dados. Em contrapartida, o CycleSort, embora seja ótimo para minimizar o número de escritas na memória, foi restringido por sua complexidade $O(n^2)$. Os testes revelaram sua inviabilidade prática para vetores com 100.000 elementos ou mais, onde falhou em concluir a ordenação, limitando seu uso a conjuntos de dados muito pequenos ou a cenários estritamente focados na economia de operações de escrita. Em conclusão, enquanto o HeapSort é a escolha robusta para desempenho em larga escala, o CycleSort é uma solução de nicho, confirmando que a seleção do algoritmo deve sempre balancear a complexidade de tempo com as restrições específicas de I/O do sistema.

Referências:

- <https://www.geeksforgeeks.org/dsa/heap-sort/>
- <https://luizcalazans.medium.com/entendendo-o-heapsort-95ec851dcdbf>
- <https://www.geeksforgeeks.org/dsa/introsort-cs-sorting-weapon/>
- <https://github.com/HamzaAhmedPasha/IntroSort--Algorithm>
- <https://share.google/mKsruGJh65THaP1ca>
- <https://www.baeldung.com/cs/cycle-sort-algorithm>
- https://youtu.be/bXwCZj1xipY?si=_8_kAnUQMSPFzI5t
- <https://youtu.be/zSYOMJ1E52A?si=H4VBWulUpUq6l3F5>
- <https://homepages.dcc.ufmg.br/~clodoveu/files/AEDS2/AEDS2.13%20Heapsort.pdf>
- <https://youtu.be/Lt4ab2GEPEo?si=9lHB0CMt54B43w1K>
- https://youtu.be/gZNOM_yMdSQ?si=VnAiJBgwOeaQoemm
- <https://techsauce.medium.com/find-the-missing-number-in-an-array-using-cycle-sort-algorithm-7de760137f8e>
- <https://cs.stackexchange.com/questions/149872/how-to-prove-cycle-sort-has-the-minimum-swap-times>
- <https://30dayscoding.com/visualizer/cyclic-sort>