Universidade Estadual Júlio de Mesquita Filho

UNESP

Projeto de Análise de Algoritmos

Análise experimental e assintótica de algoritmos de ordenação

Henrique Leal Tavares

Bauru – SP

Sumário

[1. Introdução 3](#_Toc513131066)

[2. Análise 4](#_Toc513131067)

[2.1 Bubble Sort 4](#_Toc513131068)

[2.2 Bubble Sort Otimizado 4](#_Toc513131069)

[2.3 Quick Sort 5](#_Toc513131070)

[2.3.1 Pivô Inicial 5](#_Toc513131071)

[2.3.2 Pivô Central 6](#_Toc513131072)

[2.4 Insert Sort 6](#_Toc513131073)

[2.5 Shell Sort 7](#_Toc513131074)

[2.6 Selection Sort 8](#_Toc513131075)

[2.7 Heap Sort 8](#_Toc513131076)

[2.8 Merge Sort 9](#_Toc513131077)

[3. Comparações 10](#_Toc513131078)

[3.1 Ordenação Crescente 10](#_Toc513131079)

[3.3 Ordenação Aleatória 11](#_Toc513131080)

[4. Referências 12](#_Toc513131081)

# Introdução

Os métodos de ordenação de dados são de fundamental importância em sistemas computacionais, escolher um método eficiente para determinada atividade gera um grande impacto no processamento e no desempenho de uma aplicação, existem diversos métodos famosos de ordenação, cada qual com seu potencial específico, sendo a melhor ou a pior opção em alguns cenários.

Este trabalho tem como objetivo testar, comparar e demonstrar como os algoritmos de ordenação: *BubbleSort*, *BubbleSort* com validação de lista ordenada, *QuickSort* com seu ponteiro no centro e no início do vetor, *InsertionSort*, *SelectionSort*, *ShellSort*, *MergeSort* e *HeapSort*, como eles atuam em diversos cenários de Caso médio (disposição aleatória dos vetores tentando evitar o melhor e o pior caso), Melhor caso (vetores ordenados, com a intenção de reduzir o número de comparações presentes no programa) e Pior caso (vetor disposto a entrar em todas as comparações e funções possíveis dentro do algoritmo).

Os valores dos vetores gerados foram de 1 até sua posição final, foram utilizados vetor de tamanho 1000, 5000, 10000, 15000, 20000 e 25000 posições, em ordem crescente, decrescente e aleatória, gerada pela função *random.sampe()*.

Os algoritmos foram executados no mínimo 10 vezes cada, a média de seu tempo foi calculada para uma análise mais confiável, todos foram executados nas mesmas configurações de máquina (Intel i3, placa de vídeo integrada 8GB RAM), a linguagem utilizada foi o *Python* e para captura do tempo foi utilizada a função *time.time()*.

# Análise

Nos gráficos que serão demonstrados o eixo **x** representa o valor do vetor inserido no algoritmo e o eixo **y** o tempo em **ms** (milissegundos) que o mesmo levou.

# Bubble Sort

O método de *Bubble Sort* é conhecido por sua fácil implementação e por suas complexidades semelhantes, em teoria o mesmo apenas realiza permutação do maior valor e menor valor, esse processo é repetido até que todos os itens da lista estejam em ordem.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n²) | **O(n²)** | **O(n²)** |

Figura 1 - Bubble Sort

# Bubble Sort Otimizado

Este algoritmo é muito semelhante ao *Bubble Sort* original, porém antes de realizar as permutações é executada uma verificação se o vetor já está ordenado.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n) | **O(n²)** | **O(n²)** |

Figura 2 - Bubble Sort Otimizado

# Quick Sort

O *Quick Sort* utiliza a estratégia de divisão e conquista, onde é escolhi um elemento dentro da lista (pivô) e a partir dele são permutados os maiores e melhores valores, a lista é particionada para resolução de problemas menores e a partir disto as listas são combinadas posteriormente à ordenação, para que um maior problema seja resolvido.

# Pivô Inicial

Neste caso o pivô é escolhido pelo primeiro elemento da lista.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n log n) | **O(n log n)** | **O(n²)** |

Figura 3 – Quick Sort Pivô Inicial

# Pivô Central

Neste caso o pivô é escolhido pelo elemento central da lista.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n log n) | **O(n log n)** | **O(n²)** |

Figura 4 – Quick Sort Pivô Central

# Insert Sort

*Insertion Sort* utiliza ordenação por inserção, onde dada uma lista é construída e realizada uma inserção por vez onde o objeto se encaixa da melhor forma. Pertence aos algoritmos de ordenação quadrática, descantando-se assim para organização de vetores com pequenas entradas.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n) | **O(n²)** | **O(n²)** |

Figura 5 – Insert Sort

# Shell Sort

Criado por Donald Shell em 1959 é considerado o algoritmo mais eficiente dentre os de complexidades quadráticas, é uma otimização do *Insert Short* por considerar vários seguimentos para inserção na lista, realizando diversas divisões para solução de problemas menores.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n) | **O(n²)** | **O(n²)** |

Figura 6 – Shell Sort

# 2.6 Selection Sort

O Selection Sort por sua vez tem a intenção de selecionar o menor item da lista e colocá-lo em sua posição correta, o algoritmo faz comparações de n – 1, n – 2 e assim por diante.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n²) | **O(n²)** | **O(n²)** |

Figura 7 – Select Sort

# Heap Sort

Utiliza metodologia de ordenação por seleção, criado por Robert W. Floyd em 1964 tem seus melhores resultados quando inseridos conjunto de dados ordenados de maneira aleatória.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n log n²) | **O(n log n²)** | **O(n log n²)** |

Figura 8 – Heap Sort

# 2.8 Merge Sort

O Merge Sort é outro algoritmo de ordenação que utiliza o método de divisão e conquista, divide a lista em sub-lista, realiza a ordenação das mesmas e executa a unificação destas, porém este algoritmo utiliza recursividade, o que demanda um consumo de processamento maior em comparação a outros.

|  |  |  |
| --- | --- | --- |
| Melhor caso | Caso Médio | Pior Caso |
| O(n log n²) | **O(n log n²)** | **O(n log n²)** |

Figura 9 – Merge Sort

# Comparações

Analisando gráfico a gráfico, é possível ver nitidamente a dificuldade que alguns algoritmos enfrentam conforme a entrada de dados cresce ou pelo menos muda de ordenação (crescente, aleatória e decrescente) e a diferença no processamento entre eles.

# Ordenação Crescente

Figura 10 – Vetor Ordenado

Com vetores ordenados, Bubble Sort, Quick (pivô inicial), Selection tiveram os piores resultados de maneira discrepante com relação aos outros, os demais algoritmos mostraram-se eficientes neste cenário, o Bubble Sort otimizado quase não executou instruções.

* 1. **Ordenação Decrescente**

Figura 11 – Vetor Decrescente

Para os vetores que estão em ordem decrescente os algoritmos de Bubble Sort, Bubble Sort Otimizado, Insertion Sort, Select Sort e Quick Sort apresentam os piores resultados. Quick sort com pivô central demonstrou o melhor resultado.

# Ordenação Aleatória

Figura 12 – Vetor Aleatório

A semelhança de processamento é quase a mesma em relação a vetores aleatórios e vetores decrescentes. Bubble Sort, Bubble Sort Otimizado, Insertion Sort, Select Sort e Quick Sort apresentaram os piores resultados.

# Referências

[1] Materiais dados em aula.

[2] J. E. Souza, J. V. G. Ricarte, and N. C. de Almeida Lima. Algoritmos de ordenação: Um estudo comparativo. Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA, 1, 2017.