RELATÓRIO DO PROJETO:

Implementação e Análise de Algoritmos de Ordenação em C

Introdução

O propósito desse projeto é colocar em prática os ensinamentos das aulas de algoritmos de ordenação para entender a estrutura do código e como ele funciona, bem como analisar a eficiência de cada algoritmo.

Descrição teórica dos algoritmos:

BubbleSort

O algoritmo bubble sort trabalha "borbulhando" os elementos, isso é, os elementos vão trocando de posição correta, esse algoritmo para de funcionar quando o nenhum elemento precisa ser trocado de posição, é um método que troca os elementos de lugar com frequência e é um algoritmo bom para arquivos pequenos.

SelectionSort

Esse algoritmo faz uma troca de elementos por vez, o menor (ou maior) elemento troca de posição com o último elemento da lista, desde que ele seja maior ou menor. O algoritmo para quando todos os elementos estiverem ordenados, o que, caso o arquivo já esteja ordenado, faz com que o algoritmo não ajude em nada. Esse é um algoritmo eficiente para arquivos pequenos.

InsertionSort

Esse método é utilizado quando o arquivo está quase ordenado, o custo dele é linear, ou seja, ele é eficiente quando se adiciona poucos itens a um arquivo ordenado e ele também é estável. Sua lógica é a mesma lógica utilizada pelo jogador de cartas, logo, os elementos são ordenados da esquerda para a direita um por um, então o algoritmo escolhe a posição i=1 (segundo elemento) e verifica se ele deveria estar antes ou depois da posição atual, e isso ocorre com os demais elementos até tudo estar ordenado.

MergeSort

Esse é um algoritmo do tipo dividir-para-conquistar, ou seja, o seu funcionamento é através da divisão da sequência original em pares, para, depois disso, serem ordenados e juntados no final, formando a sequência ordenada (dividiu e conquistou. No início, as partes são divididas em pares, depois agrupadas em sequências maiores até estarem completamente juntas. Então basicamente o MergeSort é dividido em 3 partes: Dividir, conquistar e combinar.

QuickSort

Esse é o algoritmo mais rápido para vários tipos de situações e, por isso, é provavelmente o algoritmo mais utilizado. Basicamente ele divide o problema em um conjunto de elementos em dois problemas menores e esses problemas menores são ordenados independentemente, por fim, no final ocorre uma combinação da solução desses problemas. O vetor em questão possui a parte da esquerda e da direita (essas partes são delimitadas através de um elemento pivô, por exemplo, a parte da esquerda vai ser: elementos <= pivô e a direita > pivô. Esse é um algoritmo que possuí uma implementação mais complicada e qualquer erro pode levar a resultados bem inesperados

HeapSort

Esse método utiliza uma estrutura de dados heap (estrutura de dados que tem árvore binária como base) para fazer a ordenação enquanto a inserção de elementos é realizada, então, no final dessa inserção, os elementos podem ser removidos dessa estrutura de forma sucessiva e ordenada. Ele é representado como um vetor ou uma árvore e, para uma ordenação crescente o seu maior elemento fica na raiz (se for decrescente, o menor fica na raiz). Para que seja heap, todos os nós devem ter o valor igual ou maior em relação aos seus filhos. Basicamente, os nós do heap podem serem relacionados em: raiz da árvore (primeira posição do vetor), filhos de um nó na posição i (posições 2i e 2i+1) e pai de um nó na posição i (posição [i/2]). É importante citar que quando acontece troca de elementos dentro da árvore, o número de trocas nunca é maior que a altura da árvore e essa troca é realizada com a remoção da raiz e a substituição dela pelo último elemento e depois ocorre as demais trocas. Então basicamente o método é feito por duas partes principais: inserir os elementos do vetor em questão em uma fila de prioridades e retirar pela raiz todos os elementos dessa fila, inserindo eles no vetor logo em seguida.

Metodologia

Para testar a performance dos algoritmos que envolve a verificação do tempo que demora para cada um solucionar a ordenação de diversos arrays com tamanhos diferentes, foi implementado no código, funções que executam os testes, foi utilizado a biblioteca <time.h> para registrar o tempo que cada algoritmo levou para terminar a operação, foi utilizado duas funções, uma que executa os testes, onde é criado um arrays que possui varios tamanhos diferentes e que serão inseridos neles valores diferentes até o seu tamanho ser preenchido, após isso será chamado cada algoritmo que ordenará esses arrays e será checado o tempo de execução desses arrays. Para

uma melhor visão dos resultados, foi criado um gráfico utilizando a ferrementa Python (matplotlib), esse gráfico estará disponível no tópico de resultados

Resultados:

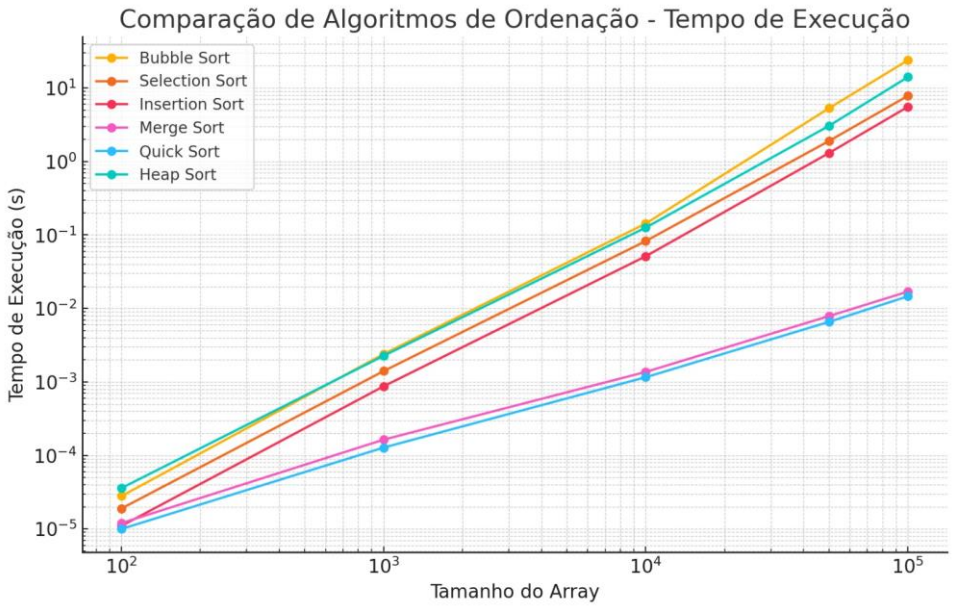
Tabela de Desempenho dos Algoritmos de Ordenação

Os testes de desempenho foram realizados para medir o tempo de execução de seis algoritmos de ordenação (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort) para diferentes tamanhos de arrays. A tabela abaixo apresenta os resultados em segundos, mostrando como o tempo de execução aumenta conforme o tamanho do array cresce.

Tamanho do Array	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort
100	0.000028	0.000019	0.000011	0.000012	0.000010	0.000036
1000	0.002399	0.001406	0.000873	0.000164	0.000128	0.002270
10000	0.143120	0.082601	0.051320	0.001369	0.001160	0.125982
50000	5.293095	1.897963	1.303259	0.007872	0.006570	3.050451
100000	23.832443	7.818599	5.533796	0.016952	0.014638	14.074942

Gráfico de Comparação dos Algoritmos de Ordenação

O gráfico a seguir ilustra o desempenho dos algoritmos de ordenação para diferentes tamanhos de arrays:



Análise de Complexidade

Selection sort

O selection sort é um algoritmo que compara um elemento com os outros a cada interação para encontrar o menor, a complexidade desse algoritmo sempre é O(n^2), ou seja, nao possui melhor caso, a sua complexidade é quadrática e é um algortimo instável e ineficiente para grandes conjuntos de dados. A sua complexidade de espaço é O(1).

Insertion sort

Esse algoritmo tem uma complexidade de espaço de O(1), o seu melhor caso é O(n) que é quando a lista já está ordenada, o seu pior caso é O(n^2) que é quando a lista está totalmente na ordem inversa e o médio é O(n^2/4) que é quando está com valores aleatórios, sem ordenação nenhuma, ou seja, não está na ordem crescente nem decrescente. Ele é um algoritmo estável e com custo linear, é bom para pequenas entradas, porém possui alto custo de movimentação dos elementos.

Bubble sort

Sua complexidade de espaço também é O(1) e possui um comportamento quadrático, seu melhor caso é O(n), mas na maioria das vezes o seu comportamento é quadrático, com o melhor e médio caso sendo O(n^2) que ocorre quando os elementos não estão ordenados, então não é bom para grandes quantidades de dados.

Merge sort

Esse algoritmo possui uma complexidade de espaço de O(n), é um algoritmo eficiente e estável, seu melhor, medio e pior caso é O(n log n) e sempre será, pois a divisão do

problema sempre gera dois subproblemas que possuem metade do problema original, ele requer espaço adicional para armazenar os elementos divididos, é um algoritmo que tende a ser menos eficiente que o quick sort, já que as suas constantes são maiores.

Quick sort

Esse algoritmo escolhe um elemento como o pivô para dividir a lista e o pior caso (O(n^2)) ocorre quando ocorre quando esse elemento pivô é o menor ou maior elemento da lista, ou seja, quando a lista ja está ordenada ou inversamente ordenada, para o melhor e medio caso O(log 2n). A complexidade de espaço é O(log n), é um algoritmo muito rapido para grandes dados, principalmente se o pivo é bem escolhido.

Heap sort

Não é um algoritmo estável mas é possivel adaptar a estrutura a ser ordenada de forma que a ordenação se torne estável, os seus casos sao O(n log n) com uma complexidade de espaço de O(1), se baseia na construção do heap a a partir de uma lista e remove o maior elemento repetidamente para que haja essa construção, tornando ele mais lento se comparado ao merge sort e quick sort

Conclusão

Neste projeto, implementamos e testamos diferentes algoritmos de ordenação para entender como funcionam e comparar a eficiência de cada um. Nos testes, vimos que algoritmos como bubble Sort, selection sort e insertion sort funcionam bem em conjuntos de dados pequenos, mas ficam muito lentos quando a quantidade de dados aumenta.

Por outro lado, algoritmos mais avançados, como merge sort, quick sort e heap sort, foram bem mais rápidos em conjuntos grandes. O quick sort foi um dos mais eficientes em muitos casos, mas sua implementação exige mais cuidado, pois depende da escolha de um elemento específico para fazer a divisão dos dados.

Com isso, vimos que a escolha do algoritmo depende do tamanho e da organização inicial dos dados. Algoritmos mais simples podem ser suficientes para dados pequenos ou quase ordenados, enquanto, em dados maiores, os algoritmos mais eficientes fazem uma diferença grande no tempo de execução. Esse projeto nos ajudou a entender melhor como cada algoritmo se comporta e a importância de escolher o método certo para cada situação.

Referências

https://pt.wikipedia.org/wiki/Selection_sort

https://pt.wikipedia.org/wiki/Insertion_sort

https://pt.wikipedia.org/wiki/Bubble_sort

https://pt.wikipedia.org/wiki/Merge_sort

https://pt.wikipedia.org/wiki/Quicksort

https://pt.wikipedia.org/wiki/Heapsort

https://pt.stackoverflow.com/questions/33319/o-que-é-a-complexidade-de-um-algoritmo

Link do Reposítorio

https://github.com/brunomartinsr/Algoritmos-de-Ordenacao