Universidade Federal de Mato Grosso do Sul

Faculdade de Computação - Sistemas de Informação

Professor: Marco Aurelio Stefanes, Algoritmos e Programação 2

Alunos: Shynji Miyasato, Ester de Lima

Relatório - Algoritmos de Ordenação

1. Descrição da atividade

O trabalho consiste em implementar, analisar e registrar o desempenho de quatro algoritmos de ordenação - *selection sort*, *insertion sort*, *merge sort* e *quicksort* - ensinados em sala, utilizando conceitos de programação como funções, ponteiros e recursividade.

Para esta atividade, foi proposta uma ordenação decrescente, onde o elemento com maior valor se encontraria na primeira posição de um vetor e, por conseguinte, o elemento com menor valor, na última posição.

2. Implementação de algoritmos de ordenação

2.1. Selection Sort

O *Selection Sort* é um algoritmo de ordenação *in-place*, isto é, que realiza a ordenação dos elementos em um conjunto de dados sem a necessidade de alocar espaço adicional para armazenar cópias dos elementos. Ele tem uma Complexidade de Tempo de $O(n^2)$ e Complexidade de Espaço de $O(1)$.

Sua ordenação é feita através da seleção do menor elemento, que é movido para o começo do vetor. Logo após isso, o segundo menor elemento é movido para o segundo índice do vetor e assim sucessivamente. O processo acaba quando o índice atinge o valor $n - 1$, pois, então, o menor elemento do vetor será obrigatoriamente ele mesmo.

Considerando que o vetor é composto por duas partes: organizado e desorganizado, o vetor começa com todos os seus elementos localizados na parte desorganizada e a parte organizada se encontra vazia. Para começar, o vetor vai ser percorrido por um índice i, partindo do primeiro índice do vetor (0) até o último ($n - 1$). Após isso, um índice j percorre o vetor partindo de i até $n - 1$. Aqui, podemos perceber que i e j percorrem a parte organizada e desorganizada do vetor, respectivamente.

O índice j vai obter o menor elemento do vetor. Caso o índice $i = j$, nada acontece, caso contrário, o conteúdo que está nos dois índices do vetor são trocados e o valor do índice i é incrementado e o processo se repete. Após concluir todas as iterações, os elementos do vetor estarão em ordem crescente.

2.2. Insertion Sort

O *Insertion Sort* é um algoritmo de ordenação *in-place* com uma estratégia de ordenação muito similar ao método utilizado para ordenar um baralho de cartas. Bem como o *Selection Sort*, o *Insertion Sort* tem uma Complexidade de Tempo de $O(n^2)$ e Complexidade de Espaço de $O(1)$.

A ordenação divide o vetor em duas partes: organizada e desorganizada, inicialmente com todos os elementos na parte desorganizada e a parte organizada vazia. A partir do primeiro item da parte desorganizada, ele é movido para a parte organizada e colocado em sua devida posição. Em seguida, o segundo item da parte desorganizada é inserido na parte organizada e movido para a esquerda até encontrar a posição correta.

Tendo um índice $i = 0$, podemos considerá-lo como já estando em sua devida posição, pois ele é o único elemento na parte organizada. Após isso, o valor do índice i é incrementado e um novo elemento da parte desorganizada é inserido na parte organizada. Se o elemento no índice $i - 1$ for maior que o elemento no índice i, eles são trocados, e isso continua até que a condição mencionada seja alcançada.

2.3. Merge Sort

O *Merge Sort* é um algoritmo de ordenação *out-of-place*, isto é, que tem a necessidade de alocar espaço adicional para armazenar cópias dos elementos para realizar a ordenação. Ele tem uma Complexidade de Tempo de $O(n \cdot log(n))$ e Complexidade de Espaço de $O(n)$.

Diferentemente dos dois últimos algoritmos, o *Merge Sort* adota uma estratégia de *Divide and Conquer* (Dividir e Conquistar), que resolve o problema em três etapas: *Divide*, onde ele quebra o problema em pedaços menores, *Conquer*, onde ele resolve cada pequeno problema, *Combine*, onde os problemas são juntados para obter o resultado final.

A ordenação divide o vetor na metade sucessivamente até que cada subproblema contenha apenas um elemento do vetor ($\frac{n}{2^k} = 1$, onde n é o tamanho do vetor e k sejam o número de divisões feitas). Em seguida, cada problema é juntado e ordenado, até que o vetor final contenha todos os elementos em ordem crescente.

2.4. Quicksort

O *Quicksort* é um algoritmo de ordenação *in-place* com a mesma estratégia adotada pelo *Merge Sort*, *Divide and Conquer*. Bem como o *Merge Sort*, possui uma Complexidade de Tempo de $O(n \cdot log(n))$, porém possui uma Complexidade de Espaço de $O(1)$, pois não precisa de alocação de espaço adicional para realizar a ordenação.

Apesar de possuírem a mesma estratégia, o *Quicksort* faz a implementação da divisão do problema de uma maneira diferente. Ao invés de separar o vetor no meio, ele escolhe um "eixo de rotação", denominado pivô, onde todos os elementos menores que o pivô se encontram à sua esquerda e todos os elementos maiores que ele se encontram à sua direita. Após isso, o processo de escolher um pivô é repetido para todos os elementos que estão à esquerda do pivô e para todos os elementos que estão à direita do pivô, separadamente.

Apesar de consideravelmente simples, o *Quicksort* tem vários pontos que podem impactar sua performance, a depender do caso. Uma das maiores problemáticas deriva da escolha do pivô. Existem alguns métodos que podem ser adotados, como o esquema de partição de Hoare, onde o primeiro elemento do vetor é escolhido como pivô, esquema de partição de Lomuto, onde o último elemento do vetor é escolhido como pivô, pivô como elemento central, pivô como mediana entre três elementos, etc.

3. Comparação de tempo de execução

Na tabela abaixo constam os tempos de execução de casos de testes aleatórios de cada um dos algoritmos. Cada linha representa os tempos de execução em segundos obtidos por um algoritmo de ordenação para um dado tamanho de entrada.

Todos os tamanhos de entrada foram informados manualmente, mas todos os valores que foram utilizados para preencher o vetor durante a execução foram escolhidos de forma aleatória e automática.

	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Selection	0.000003	0.000003	0.000072	0.007344	0.704891	54.43205	5443.21	544321	54432100
Insertion	0.000003	0.000003	0.000042	0.003015	0.380245	40.24325	4024.33	402433	40243300
Merge	0.000003	0.000003	0.000021	0.000150	0.001533	0.018874	0.276883	3.165553	32.73942
Quick	0.000003	0.000003	0.000014	0.000109	0.001193	0.016557	0.155119	2.023378	20.53790

Os casos de teste para 10^6+ para os algoritmos *Selection Sort* e *Insertion Sort* não puderam ser executados por dificuldades técnicas. Portanto, os tempos de execução são apenas projeções matemáticas.

4. Conclusão

Através da Análise Assintótica, conseguimos calcular o desempenho de algoritmos de forma objetiva e definir um Limite Superior Assintótico para o Tempo de Execução. No entanto, é importante notar que algoritmos com a mesma Complexidade de Tempo podem ainda apresentar diferenças significativas no Tempo de Execução, devido a diversos fatores, incluindo a Complexidade de Espaço e a escolha de métodos específicos.

No geral, esta pesquisa demonstrou que o *Quicksort* apresenta o melhor desempenho para casos com mais de 10 elementos. Entretanto, é importante destacar que sua implementação pode ser mais complexa em comparação com outros algoritmos. Portanto, ao escolher o melhor algoritmo de ordenação, é essencial considerar tanto a complexidade quanto o desempenho, levando em conta o contexto e os objetivos. Por isso, algoritmos como o *Bubble Sort* podem ser valiosos para estudantes que estão aprendendo sobre análise e compreensão de algoritmos, enquanto o *Quicksort* pode ser a escolha preferencial para ordenar listas de registros em um SGBD.