Relatório do Projeto 1 - PAA

Flávia Narumi Nitto

25 de outubro de 2021

Resumo

O relatório tem como principal objetivo abordar o problema da ordenação e os diferentes comportamentos que cada algoritmo implementado assume sob diferentes valores de entrada de tamanhos variados. Nessa linha de raciocínio, foi feita a coleta do tempo de execução que cada método levou e os dados foram apresentados em tabelas. Além disso, discutimos os resultados obtidos e comparamos o desempenho dos algoritmos entre si. Então, concluímos o estudo proposto pelo projeto analisando os resultados tabelados com o que foi estudado na teoria: qual situação era o melhor e o pior caso, qual algoritmo foi o mais eficiente e o menos eficiente e quais os contextos mais adequados para a implementação de cada método.

1 Introdução

O problema da ordenação consiste em definir qual o melhor algoritmo a ser implementado dependendo da situação problema. No entanto, para que possamos definir de fato qual o mais eficiente, é preciso considerar vários fatores, tais como conhecimento prévio das chaves, hierarquia de memória do computador, software sendo utilizado e vários outros. [1]

Nesse contexto, com o avanço tecnológico os computadores ficaram consideravelmente mais rápidos se comparados com os da década passada, no entanto eles não são infinitamente rápidos e nem têm memória ilimitada. Dessa maneira, devido à memória limitada das máquinas, fica evidente a importância da análise e projeto de algoritmos, pois é preciso um estudo para otimizar os recursos de tempo e espaço a serem utilizados.

Pensando nesse problema, o projeto foi desenvolvido com o objetivo de ilustrar de forma prática como diferentes algoritmos desenvolvidos para o mesmo propósito (de ordenar vetores) se comportam com valores de entrada de 1.000, 10.000 e 50.000 posições, sejam eles ordenados de forma crescente, decrescente e completamente desordenados.

A título de conclusão, foi constatado que embora os algoritmos tenham a mesma finalidade de ordenar vetores, cada um possui diferentes complexidades, utiliza diferentes estruturas de dados e ocupa uma determinada quantidade de memória. Isso reforça a ideia de que para escolher o melhor algoritmo, há inúmeros fatores a serem considerados. Por isso, a questão da análise e projeto de algoritmos é tão importante na atual conjuntura, com ela podemos estudar o caso e então aplicar o método mais eficiente. Por fim, depois das testagens e comparações, pudemos confirmar que a teoria está, de fato, em consonância com o que foi testado na prática.

2 Fundamentação Teórica

Para a elaboração desse trabalho, foram necessários conhecimentos nas linguagens C e C++, bem como a interpretação e análise das diferentes estruturas de dados utilizadas pelos algoritmos de

ordenação. Um exemplo disso é o constante uso e manipulação de ponteiros, bem como a questão do alocamento de memória dinâmica para os vetores.

2.1 *Insertion sort*

O *Insertion sort* é um algoritmo estável de comparação, usado principalmente em vetores de tamanho pequeno em que há poucos elementos a serem ordenados.

Funcionamento
Seu funcionamento consiste em posicionar um elemento em seu devido lugar a cada iteração, tem comportamento bem semelhante ao de ordenar as cartas em nossa mão em um jogo de cartas. Ou seja, assumimos que o primeiro elemento do vetor está ordenado e armazenamos o valor do segundo elemento em uma chave. Em seguida, comparamos o valor presente na chave com o primeiro elemento. Se o primeiro elemento for maior que o valor da chave, então o valor da chave é posicionado à direta do primeiro elemento, caso contrário, o valor é posicionado à esquerda. Repetimos esse processo até que todos os elementos estejam no seu devido lugar. [1]

Análise de Complexidade

- Pior caso: $O(n^2)$
  Ocorre quando temos um vetor ordenado de forma crescente e queremos ordená-lo de forma decrescente. Isso faz com que cada elemento seja comparado com todos os outros n elementos $(n-1)$ vezes.
- Melhor caso: $\Omega(n)$
  Ocorre quando o vetor já está ordenado, então o laço de repetição externo é executado apenas uma vez e o laço interno não é executado. Ordem de complexidade é linear pois são feitas apenas n comparações.
- Caso médio: $\Theta(n^2)$
  Ocorre quando os elementos do vetor estão embaralhados de forma aleatória (nem crescente, nem decrescente).

2.2 *Shell sort*

O método de ordenação *Shell sort* é um algoritmo não estável de comparação muito utilizado em casos específicos. Como por exemplo quando um algoritmo excede o limite de recursões. Outra aplicação é encontrada para resolver o problema de quando o *Insertion sort* não performa bem (pois seus elementos com valor próximo estão muito distantes), com o *Shell sort* essa distância entre os elementos de valor próximo é reduzida e então, menos trocas serão realizadas.

Há vários métodos de implementação para esse algoritmo com diferentes performances, mas para o trabalho foi escolhido o método Knuth que define o passo com a fórmula a seguir

$$passo = \frac{3^k - 1}{2}$$

Funcionamento
A principal característica do *Shell sort* é que o algoritmo compara elementos que estão distantes uns dos outros no vetor, por esse motivo, os elementos menores são movidos rapidamente para o final do vetor. Dessa forma, é evidente que o vetor não poderá ser ordenado em apenas uma execução como era feito no *Insertion sort*, então, a estratégia é utilizar um determinado **passo**

para as primeiras comparações e ir diminuindo seu valor conforme as iterações. [2]

Análise de Complexidade
A análise de complexidade do *Shell sort* é algo que não pode ser definido exatamente, devido ao fato do algoritmo se comportar de diferentes maneiras com diferentes passos. Outra questão a ser considerada é o vetor com elementos de entrada, pois dependendo de como os valores estão espalhados pela estrutura a ordem de complexidade é alterada. No entanto, há conjecturas referentes à notação assintótica da sequência de Knuth:

- Conjectura 1: $\Theta(n^{1.25})$
- Conjectura 2: $\Theta(n(log\ n)^2)$

2.3 *Merge sort*

O *Merge sort* é um algoritmo estável de comparação que utiliza o princípio da divisão e conquista, além disso, é muito utilizado para resolver problemas de contagem de inversões (determinar o quão longe o vetor está de ser um vetor ordenado).

Funcionamento
O algoritmo divide recursivamente o vetor em duas metades até que tenhamos subproblemas de tamanho 1. Depois disso, a função *merge()* é executada e combina todos os subproblemas já ordenados até que o vetor inteiro esteja ordenado.

Análise de Complexidade
Um ponto interessante do *Merge sort* é que, independentemente dos valores de entrada, ele será executado $(n\ log\ n)$ vezes. Portanto, não há pior caso nem melhor caso. [2]

- Pior caso: $O(n\ log\ n)$
- Melhor caso: $\Omega(n\ log\ n)$
- Caso médio: $\Theta(n\ log\ n)$

2.4 *Quick sort*

O método de ordenação *Quick sort* é um algoritmo não estável de comparação que utiliza o princípio da divisão e conquista, além disso, é geralmente implementado em programas que prezam pela otimização do tempo de execução.

Funcionamento
O algoritmo divide o vetor em subproblemas a partir da seleção de um elemento pivô. Durante essa divisão, o elemento pivô deve ser posicionado de uma forma que os elementos menores que ele sejam posicionados à sua esquerda e os maiores à sua direita. O algoritmo implementado no projeto define o elemento pivô como sendo o elemento presente na última posição do vetor. Após a primeira iteração, dividimos os vetores da esquerda e da direita recursivamente utilizando essa mesma técnica até que os subvetores tenham tamanho 1. Com isso, todos os elementos já estão ordenados e basta combiná-los no vetor original.

Análise de Complexidade

- Pior caso: $O(n^2)$
  Ocorre quando o elemento selecionado para ser o pivô possui o maior (ou menor) valor

do vetor. Isso faz com que o pivô fique na extremidade do vetor ordenado, ou seja, um subproblema sempre será vazio e o outro irá conter todos os outros $(n-1)$ valores do vetor. Então a função *quicksort()* só será chamada no vetor com $(n-1)$ elementos.

- Melhor caso: $\Omega(n\ log\ n)$
  Ocorre quando o elemento selecionado para ser o pivô é o elemento do meio, ou bem próximo do elemento do meio. Com isso, os subproblemas terão tamanhos balanceados e a função *quicksort()* será chamada igualmente nos subvetores.
- Caso médio: $\Theta(n\ log\ n)$
  Ocorre quando o elemento selecionado para ser o pivô não cumpre os requisitos do pior caso e do melhor caso. Ou seja, os subproblemas são divididos de forma relativamente balanceada.

2.5 *Bucket sort*

O *Bucket sort* é um algoritmo de ordenação estável de distribuição que utiliza o conceito de *scatter-gather*, é aplicado quando o vetor de entrada tem seus valores distribuídos uniformemente em um intervalo.

Funcionamento
O vetor é dividido em vários grupos, chamados de buckets(baldes). Dessa forma, após separar os elementos nos buckets, eles são ordenados usando um método estável de ordenação. No nosso algoritmo, usamos o método de ordenação *Insertion sort*. Por fim ocorre a combinação de todos os elementos ordenados, onde os valores são copiados de volta para o vetor original e apagados dos buckets.

Análise de Complexidade

- Pior caso: $O(n^2)$
  Ocorre quando os elementos do vetor estão ordenados em ordem reversa e utilizamos o *Insertion sort* para ordenar os elementos de dentro do bucket.
- Melhor caso: $\Omega(n+k)$
  Ocorre quando os elementos estão uniformemente distribuídos nos buckets com uma quantidade parecida de elementos em cada. No caso de os elementos dentro dos buckets já estarem ordenados, a complexidade fica melhor ainda.
  Se usarmos o algoritmo *Insertion sort* para ordenar os elementos dentro dos buckets, então teremos complexidade $\Omega(n+k)$, onde $\Omega(n)$ é a complexidade para formar os buckets e $\Omega(k)$ é a complexidade para ordenar os elementos do bucket usando algoritmos de ordem linear no melhor caso.
- Caso médio: $\Theta(n)$
  Ocorre quando os elementos estão aleatóriamente espalhados pelo vetor. Embora os elementos não estejam distribuídos uniformemente, o algoritmo ainda se comporta na complexidade linear.

2.6 *Counting sort*

O *Counting sort* é um algoritmo de ordenação de distribuição estável muito utilizado em casos que a complexidade linear é necessária. Outra aplicação recorrente do *Counting sort* é quando estamos trabalhando com um vetor de inteiros com múltiplas contagens.

Funcionamento
O funcionamento do algoritmo consiste em ordenar os elementos do vetor contando o número de ocorrências de cada elemento do vetor. Essa contagem é armazenada em um vetor auxiliar e a ordenação é feita utilizando o valor da contagem como índice no vetor auxiliar.

Análise de Complexidade
Em todos os casos, a complexidade do *Counting sort* é a mesma, pois independentemente dos valores de entrada o algoritmo é executado $(n + k)$ vezes. No entanto, um exemplo de entrada ruim é quando os valores inteiros são muito grandes, com isso o vetor auxiliar precisará ser do tamanho do maior elemento do vetor principal.

- Pior caso: $O(n + k)$
- Melhor caso: $\Omega(n + k)$
- Caso médio: $\Theta(n + k)$

3 Metodologia

Todos os algoritmos utilizados foram implementados na linguagem C/C++ e a principal estrutura de dados utilizada foi o vetor. Além disso, para a implementação do algoritmo *Bucket sort* foi necessário o uso da estrura de dados de uma lista encadeada para armazenar os elementos nos devidos *buckets*.

Outro ponto importante sobre o *Bucket sort* é que durante o período de testagem, foi preciso ajustar o número de *buckets* e o intervalo de cada *bucket* de acordo com o tamanho da entrada.

Tamanho	Nº Buckets	Intervalo
1.000 posições	10	100
10.000 posições	100	100
50.000 posições	500	100

Tabela 1: Adaptações necessárias no algoritmo Bucket Sort

A função *clock_gettime()* da biblioteca *time.h* foi utilizada com o propósito de calcular quanto tempo o algoritmo de ordenação levou para concluir toda a execução, essa função retorna o valor em segundos com cerca de 6 casas decimais de precisão.

O serviço utilizado foi o terminal gcc do Visual Studio Code em conjunto com o terminal Linux para a fase final de testes. O hardware utilizado em todas as etapas da construção, implementação e testagem dos algoritmos foi uma máquina com sistema operacional de 64 bits, 8GB de memória RAM e processador Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz.

4 Resultados

Para a fase de testes, foram escolhidos três tamanhos de vetores de entrada e salvos em arquivos de texto .dat para facilitar a execução dos programas. Nesse período, testamos vetores de 1.000, 10.000 e 50.000 posições com os valores ordenados de forma crescente, decrescente e desordenados. Após a testagem e medição dos tempos de execução dos algoritmos de ordenação com diferentes tamanhos de entrada, obtemos os seguintes resultados:

Insertion sort			
Tamanho	Crescente	Decrescente	Desordenado
1.000 posições	0.001139 seg	0.023481 seg	0.015372 seg
10.000 posições	0.011589 seg	0.165249 seg	0.046648 seg
50.000 posições	0.026521 seg	0.125898 seg	1.260305 seg

Tabela 2: Desempenho do Insertion Sort

Shell sort			
Tamanho	Crescente	Decrescente	Desordenado
1.000 posições	0.01447 seg	0.02634 seg	0.02211 seg
10.000 posições	0.01612 seg	0.03691 seg	0.02335 seg
50.000 posições	0.03067 seg	0.04846 seg	0.03775 seg

Tabela 3: Desempenho do Shell Sort

Merge sort			
Tamanho	Crescente	Decrescente	Desordenado
1.000 posições	0.01675 seg	0.01624 seg	0.01938 seg
10.000 posições	0.02194 seg	0.02369 seg	0.02048 seg
50.000 posições	0.04169 seg	0.03323 seg	0.03664 seg

Tabela 4: Desempenho do Merge Sort

Quick sort			
Tamanho	Crescente	Decrescente	Desordenado
1.000 posições	0.01645 seg	0.01529 seg	0.01255 seg
10.000 posições	0.14064 seg	0.12474 seg	0.02572 seg
50.000 posições	4.97523 seg	0.55689 seg	0.03558 seg

Tabela 5: Desempenho do Quick Sort

Bucket sort			
Tamanho	Crescente	Decrescente	Desordenado
1.000 posições	0.01296 seg	0.02405 seg	0.01723 seg
10.000 posições	0.01958 seg	0.02933 seg	0.03123 seg
50.000 posições	0.02621 seg	0.05155 seg	0.05451 seg

Tabela 6: Desempenho do Bucket Sort

Counting sort			
Tamanho	Crescente	Decrescente	Desordenado
1.000 posições	0.02464 seg	0.02056 seg	0.02099 seg
10.000 posições	0.03911 seg	0.02291 seg	0.02401 seg
50.000 posições	0.04344 seg	0.06653 seg	0.03248 seg

Tabela 7: Desempenho do Counting Sort

5 Discussão

Com base nos resultados apresentados, pode-se dizer que de fato, a teoria está de acordo com o que foi posto em prática. Nesse contexto, comparando o comportamento dos diferentes tipos de algoritmos de ordenação para os diferentes tipos e tamanhos de entrada, podemos averiguar o melhor caso dos métodos *Insertion* e *Bucket* quando o vetor já está ordenado (em ordem crescente), como proposto pela teoria estudada. Dessa maneira, também é possível constatar que o pior caso dos métodos *Insertion* e *Bucket* ocorre quando o vetor de entrada está ordenado em ordem decrescente, como podemos ver pela tabela 2 e 6.

O método *Quick sort*, por sua vez, tem o pior caso quando o vetor de entrada está ordenado de forma crescente ou de forma decrescente, como ilustrado na tabela 5, nela podemos observar que o algoritmo demora quase 5 segundos para ordenar o vetor de 50.000 posições ordenado de forma crescente. Em contrapartida, os algoritmos *Shell*, *Merge* e *Counting* não são afetados pelos valores de entrada, portanto, possuem tempos de execução semelhantes em todos os casos de entradas testadas.

Contexto mais adequado

- Insertion sort: Quando temos um vetor com poucos elementos desordenados.
- Shell sort: Adequado quando o *Insertion sort* não performa bem, ou seja, quando os elementos de valor próximo estão muito distantes no vetor.
- Merge sort: Ideal para casos em que é necessário ordenar listas ligadas.
- Quick sort: Eficiente quando precisa-se de um algoritmo de ordenação rápido, estável e que não precise de memória adicional.
- Bucket sort: Adequado para situações onde os elementos estão uniformemente distribuidos em um determinado intervalo.
- Counting sort: Quando a complexidade linear é prioridade e os valores inteiros não são muito altos (senão o vetor auxiliar acaba ocupando muita memória).

Ademais, foi observado que o desempenho do algoritmo *Bucket sort* depende de como o número de buckets e o intervalo foi definido, pois podemos definir as métricas de forma que o algoritmo fique extremamente lento e ineficiente. Um exemplo disso é quando o número de baldes e o intervalo é muito grande, isso ocasiona em um desperdício de recursos de memória, pois todos os elementos de entrada ficam apenas em um balde e todos os outros baldes ficam vazios.

6 Conclusões

Portanto, partindo dos resultados obtidos através dos testes com vetores de entrada de 1.000, 10.000 e 50.000 posições, sejam eles ordenados de forma crescente, decrescente e com valores desordenados. Pode-se concluir que o *Insertion*, *Shell* e *Bucket* são os algoritmos mais rápidos para o caso em que o vetor de entrada já está ordenado crescentemente. Em contrapartida, quando nos referimos aos métodos de ordenação com os valores de entrada ordenados de forma decrescente, temos que o *Insertion* e o *Quick* são os mais lentos comparados com os outros algoritmos. Finalmente, quando os elementos de entrada estão desordenados utilizando a função *rand()* da linguagem C, observa-se que há uma grande possibilidade de termos atingido o caso médio na maioria dos algoritmos.

Referências

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Robert Sedgewick and Kevin Wayne. *Algorithms, Fourth Edition*. Person Education, 2011.