Trabalho Prático 2: Biblioteca Digital de Arendelle

Victor Hugo Silva Moura

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

victorhugosmoura@gmail.com

Abstract. *The following text describes the "Arendelle's Digital Library" problem and its solution using different implementations of the QuickSort algorithm. For each implementation, a series of tests were conducted in order to analyze and compare their respective performances.*

Resumo. *O texto a seguir descreve o problema "Biblioteca Digital de Arendelle" e sua resolução usando diferentes implementações do algoritmo QuickSort. Para cada implementação é feita uma série de testes a fim de analisar e comparar suas respectivas performances.*

1. Introdução

O problema *Biblioteca Digital de Arendelle* consiste em ordenar o acervo de livros e pergaminhos que o reino de Arendelle possui em suas bibliotecas físicas. Por simplificação, cada item do acervo é considerado como um número inteiro e, deste modo, o real problema é ordenar um array de inteiros. Para realizar essa ordenação existem 7 algoritmos candidatos que devem ser comparados a fim de descobrir qual é a eficiência deles para a resolução desse problema. Os algoritmos candidatos são variações do algoritmo QuickSort, sendo eles:

1. **Quicksort clássico** - Seleção de pivô usando o elemento central.
2. **Quicksort mediana de três** - Seleção do pivô usando a "mediana de três" elementos, em que o pivô é escolhido usando a mediana entre a chave mais à esquerda, a chave mais à direita e a chave central (como no algoritmo clássico).
3. **Quicksort primeiro elemento** - Seleção do pivô como sendo o primeiro elemento do subconjunto.
4. **Quicksort inserção 1%** - O processo de partição é interrompido quando o subvetor tiver menos de k = 1% chaves. A partição então deve ser ordenada usando uma implementação especial do algoritmo de ordenação por inserção, preparada para ordenar um subvetor. Seleção de pivô usando a "mediana de três" elementos, descrita acima.
5. **Quicksort inserção 5%** - Mesmo que o anterior, com k = 5%.
6. **Quicksort inserção 10%** - Mesmo que o anterior, com k = 10%.
7. **Quicksort não recursivo** - Implementação que não usa recursividade. Utiliza pilha para simular as chamadas de função recursivas e identificar os intervalos a serem ordenados a cada momento. A seleção do pivô deve ser feita assim como no Quicksort clássico.

Para cada algoritmo são feitos testes com diferentes tipos de arrays de entrada (ordenado crescente, ordenado decrescente e aleatório) e diferentes tamanhos de array com intuito de obter aproximadamente o tempo de execução, número de comparações entre chaves e número de movimentações de registro médios (mediana no caso do

tempo de execução). Posteriormente deve ser feita uma análise dos três parâmetros citados anteriormente. A linguagem utilizada na solução desse problema foi a linguagem C.

2. Implementação

A implementação desse problema se deu em duas etapas distintas, sendo elas:

- Identificar os parâmetros passados para o programa (tipo de QuickSort, tamanho e organização dos arrays, impressão dos arrays) e organizar os dados seguindo esses parâmetros;
- Montar e executar o tipo certo de QuickSort de acordo com a entrada do programa.

Sendo assim, vamos dividir essa seção em duas partes para abordar cada etapa individualmente.

2.1. Identificação dos parâmetros

Os parâmetros para o programa são passados na forma de argumentos no seguinte formato:

./nomedoprograma <variação> <tipo> <tamanho> [-p]

onde *variação, tipo* e *tamanho* são o tipo de QuickSort a ser executado, o tipo de ordenação dos arrays e o tamanho do arrays, respectivamente. O parâmetro -p, quando presente, indica que é necessário imprimir os arrays originais utilizados na ordenação.

A linguagem C fornece uma forma simples de lidar com argumentos. Por meio da passagem de dois parâmetros para a função main, é possível saber a quantidade de argumentos e seus valores. Esses parâmetros são o *argc* e o *argv*, sendo o primeiro um valor inteiro que guarda a quantidade de argumentos e o segundo um array de strings contendo o valor dos argumentos. Dessa forma, a declaração do main fica no seguinte formato:

*int main(int argc, char *argv[]) { … }*

Após o recebimento dos argumentos/parâmetros, a variação e o tamanho dos arrays são passados para uma função responsável por gerar os arrays de teste. Os arrays de ordem aleatória utilizam um função que gera números aleatórios (*rand()* da biblioteca stdlib.h) enquanto que os arrays ordenados em ordem crescente e decrescente vão de 1 a n e de n a 1, respectivamente, sendo n o tamanho dos arrays.

Imediatamente após gerar os arrays, é conferido se o parâmetro -p estava presente na entrada. Se sim, é feita uma cópia dos arrays originais de forma a garantir que a ordem original de cada um deles não seja perdida. Por fim, é feita uma chamada para a função QuickSort indicando o tipo de ordenação que foi passado. Esta função, que será abordada na próxima seção, é responsável por identificar e aplicar o tipo certo de ordenação para os arrays.

2.2. QuickSort

Inicialmente cada variação de QuickSort foi tratada de forma individual, sendo criado um conjunto específico de funções para cada uma delas. Porém, após desenvolver e testar cada variação, o objetivo se tornou reduzir a quantidade de código repetido por meio da junção de algumas funções. Com esse processo, as variações de QuickSort

foram divididas em três grupos principais: QuickSort Padrão, QuickSort com Inserção e QuickSort Não Recursivo. O primeiro grupo é composto pelas variações QuickSort Clássico, Mediana de Três e Primeiro Elemento, cuja variação se dá apenas pela escolha do pivô. O segundo grupo é composto por QuickSort Inserção 1%, 5% e 10%, cuja variação se dá apenas pela quantidade de elementos necessários para a chamada do método de ordenação por inserção. O último grupo é composto apenas pela variação QuickSort Não Recursivo, pois essa é a única variação que não utiliza recursão.

Para cada um dos dois primeiros conjuntos foi desenvolvida uma função Sort que coordena a escolha de pivô e as chamadas recursivas (além da chamada para o método de inserção, no caso do segundo grupo). O método de partição é único no código, ou seja, todas as variações de QuickSort utilizam ele, e a diferenciação entre os formas de escolha de pivô é indicada por um parâmetro passado para a função. Uma estrutura de dados Pilha é utilizada na função de QuickSort Não Recursivo para armazenar a ordem das chamadas para a função de partição, guardando os indíces do array para as próximas chamadas.

As funções Sort dos grupos QuickSort Padrão e QuickSort com Inserção, e a função de QuickSort não recursivo são coordenadas por uma função geral chamada QuickSort (mencionada no final da seção anterior). Essa função, que recebe um array e sua variação, determina qual função de ordenação será chamada e quais parâmetros serão passados para a função. Por exemplo: para a variação QuickSort Clássico, o QuickSort chama a função Sort do grupo QuickSort Padrão passando como parâmetro de escolha de pivô o número 0, que indica que o pivô escolhido será o elemento central. Para um QuickSort Mediana de Três, a chamada seria a mesma, com exceção do parâmetro de escolha de pivô que agora seria o número 2, indicando uma escolha de pivô por mediana de três. Para o grupo de QuickSort com Inserção, é passado o número de elementos a partir do qual o método de inserção deve ser chamado.

As funções de inserção e partição possuem um contador de trocas de elementos do vetor e de comparações entre elementos do vetor que são retornadas a cada chamada. Isso possibilita a contagem desses parâmetros, que será necessária na saída do programa e na seção de Análise Experimental (seção 4). A saída do programa segue o seguinte formato:

<variação> <tipo> <tamanho> <n_comp> <n_mov> <tempo>

onde *variação*, *tipo* e *tamanho* são os parâmetros recebidos na entrada, *n_comp* e *n_mov* se referem ao número médio de comparações de chaves e de movimentações de registros efetuadas e *tempo* ao tempo mediano de execução, em microssegundos.

3. Instruções de Compilação e Execução

O compilador utilizado durante toda a implementação foi o GNU C Compiler (GCC) versão 7.3.0 e o sistema operacional foi um Linux Ubuntu 18.04 64-bit.

Para compilar o programa foi desenvolvido um arquivo makefile que é responsável por gerenciar os arquivos a serem compilados e a ordem de compilação. Para executar esse aqui, basta digitar o comando *make* no terminal.

A execução do programa (como mostrado na seção 2.1) é feita pelo comando:

./nomedoprograma <variação> <tipo> <tamanho> [-p]

Obs.: Para alguns casos de teste, o número de chamadas recursivas ultrapassa o limite da pilha de execução devido ao grande número de elementos somado ao pior caso do QuickSort. Para evitar isso, o comando *ulimit -s hard*, que aumenta o tamanho pilha, deve ser usado antes da execução do programa.

4. Análise Experimental

As análises experimentais seguiram um modelo determinado pelas instruções do trabalho. O modelo consiste em fazer testes de arrays variando entre 50 mil e 500 mil elementos, em intervalos de 50 mil elementos, para cada variação de QuickSort e de ordenação do array. Sendo assim temos: *10 tamanhos* x *7 quicksorts* x *3 ordenações do array* = 210 testes. Porém, para cada teste foram usados 20 arrays de modo a obter a mediana do tempo de execução e, as médias de comparações e movimentações de elementos no array. Sendo assim, *4200* arrays foram durantes os testes.

Especificações técnicas do computador de testes:

- **Processador:**
- **Memória Ram:**
- **Sistema Operacional:** Linux Ubuntu (versão 18.04)

Os dados utilizados para análise e todos os gráficos estão na pasta *dados* enviada junto com esta documentação. Os arquivos dos gráficos estão nomeados de forma a facilitar a identificação de cada um deles.

4.1. Análise do Tempo

A sub-pasta *Time* contém todos os gráficos que comparam a performance (em tempo) dos métodos de ordenação para determinada ordenação inicial dos elementos e quantidade de elemtentos do array. Entre eles, dois gráficos são bem interessantes de serem analisados. O primeiro (Figura 1) compara a mediana do tempo para arrays de 500000 elementos ordenados previamente em ordem crescente. O gráfico à esquerda mostra os dados em escala real, enquanto que o gráfico à direita mostra um zoom do primeiro gráfico, o que possibilita ver o tempo de todos os métodos.

O que pode-se notar por meio desses gráficos é que o tempo de ordenação do QuickSort com pivô no primeiro elemento é consideravelmente maior do que o dos outros métodos. Isso ocorre porque como os dados já estão ordenados, na hora da partição o pivô sempre será o menor elemento do sub-vetor a ser ordenado, gerando assim a pior partição possível para os elementos. Assim, o tempo de ordenação por esse método entra no pior caso do QuickSort, que é $O(n^2)$.

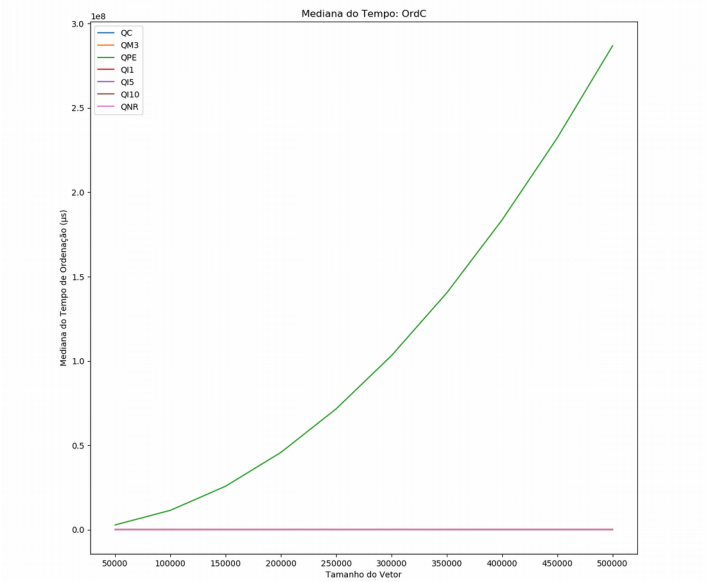

Figura 1. Comparação do tempo para vetores ordenados

5. Conclusão

6. Referências