Trabalho Prático 2
Biblioteca Digital de Arendelle

Lucas Paulo Martins Mariz

2018055016

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

lucaspaulom@hotmail.com

Abstract: *The following article describes in a precise, succinct and qualitative way the implementation and solution of the problem proposed as a practical work of the Data Structure discipline. In short, the text derives through the different implementations of quicksort, its characteristics, efficiency, performance, among other aspects presented with the aid of tables and graphs. The median of the execution time, the average of the number of comparisons as well as the average number of movements will be considered as objects of analysis.*

Resumo: *O seguinte artigo descreve de forma precisa, suscinta e qualitativa a implementação e solução do problema proposto como trabalho prático da disciplina Estrutura de Dados. Em suma, o texto decorre através das diferentes implementações de quicksort, suas características, eficiência, performance, entre outros aspectos apresentados com o auxílio de tabelas e gráficos. Serão considerados como objetos de análise a mediana do tempo de execução, a média do número de comparações bem como a média do número de movimentações.*

1. Introdução

Em um contexto geral, foi apresentado o problema de uma grande biblioteca e a sua dificuldade de lidar com a falta de ordenação de sua grande quantidade de livros. Felizmente, cada exemplar é identificado por um número ao qual atribuiremos a função de identificador, considerando, obviamente, a sua unicidade. Sendo assim, devemos implementar diferentes soluções com princípios bem próximos com o objetivo de analisar a eficiência de cada implementação. Dessa forma, são descritas abaixo os tipos de quicksort a serem decorridos:

1. **Quicksort Clássico (QC):** o elemento central é escolhido como pivô;
2. **Quicksort Mediana de três (QM3):** a seleção do pivô é realizada através da mediana de três entre o primeiro elemento, o elemento central e o último;
3. **Quicksort Primeiro Elemento (QPE):** o primeiro elemento é selecionado como pivô.
4. **Quicksort Inserção 1% (QI1):** o quicksort é utilizado até que sobre 1% dos elementos para ordenar. Dessa forma, o processo de partição é interrompido e inicia-se a utilização do insertion sort para ordenar o restante. O pivô, assim como o anterior, é selecionado através da mediana de 3;

5. **Quicksort Inserção 5% (QI5):** mesmo que o anterior, tendo a condição de parada em 5%;
6. **Quicksort Inserção 10% (QI10):** mesmo que o anterior, tendo a condição de parada em 10%;
7. **Quicksort Não Recursivo (QNR):** implementação que não usa recursividade. Para tal, será utilizada uma pilha (*stack*) para poder "simular" essas chamadas. O elemento central será denominado como pivô.

Por motivos de simplificação, utilizaremos as siglas determinadas acima para referir a cada uma das implementações mencionadas.

Devido ao grande número de livros, poderemos recebê-los de várias formas, cabendo ao algoritmo estar preparado para lidar com essas possíveis diferentes situações. Mais especificamente, os dados podem vir ordenada de forma crescente, ordenada de forma decrescente ou aleatória (cabendo ao programa gerar tais dados).

Além disso, podemos receber diferentes quantidades de livros, sendo que esse número varia de 50 mil em 50 mil, tendo como limite inferior 50 mil e limite superior de 500.000 elementos. Serão analisados a mediana do tempo de execução, a média do número de comparações bem como a média do número de movimentações com o auxílio de tabelas e gráficos.

2. Implementação

O programa foi desenvolvido na linguagem C, compilado pelo GCC versão 7.4.0 em um sistema operacional Ubuntu 18.04. Todo o código foi executado em meu notebook cujas especificações principais estão descritas abaixo:

- Intel Core i5-7200U CPU 2.5GHZ x 4
- Nvidia G-Force GTX 840M
- 8GB RAM
- SSD Kingston A400 240GB
- Ubuntu 18.04.2 LTS

Primeiramente será feito uma breve introdução ao algoritmo de ordenação quicksort e logo após será apresentado os métodos utilizados para sua implementação.

2.1. Quicksort

O quicksort é um dos algoritmos de ordenação mais rápidos da atualidade, possuindo uma complexidade de O(nlog(n)) para grande parte dos casos. Sua implementação segue a seguinte lógica:

1. Escolhe o pivô;
2. Rearranja o vetor de forma que todos os elementos anteriores ao pivô sejam menores que ele e, de forma análoga, todos os elementos posteriores ao pivô sejam maiores que ele. Por fim, o pivô chegará em sua posição e haverá sub vetores não ordenados (partição).
3. Repetimos os passos anteriores de forma recursiva até que todos os sub vetores sejam ordenados.

Como explicado na introdução, foram implementadas 3 formas diferentes de seleção de pivô:

- Escolha do elemento central como pivô *(QC e QNR)*.
- Escolha da mediana de três entre o primeiro, central, ultimo elementos como pivô (*QM3, QI1, QI5 e QI10)*.
- Escolha do elemento primeiro elemento como pivô no *QPE*.

Dessa forma, o passo 1 da lógica de quicksort é bem dinâmico uma vez que todo o fluxo de ordenação pode ser alterado ao trocar o método de escolha do pivô.

O quicksort inserção, por sua vez, possui um diferencial em seu método de ordenação. De forma suscinta, o quicksort é utilizado normalmente até que certa condição de parada seja atingida. Em outras palavras, uma certa porcentagem do vetor é ordenada e as partições restantes são destinadas ao método de inserção. A condição de parada se refere a porcentagem do tamanho do vetor que deve ser destinada ao *insertion sort,* valor que pode ser 1%, 5% e 10%.

Com o intuito de modularizar o código, a mesma função de quicksort foi utilizada para a maioria das implementações, menos para o caso recursivo. Primeiramente uma espécie de interface organiza os parâmetros de cada tipo de ordenação, vetor e forma de escolha do pivô. Em relação à forma de escolher o pivô, o controle é feito através de um inteiro que, dependendo do seu valor, seleciona os diferentes tipos de pivô. São eles:

- tipo = 0 => o pivô é o elemento do meio;
- tipo = 1 => o pivô é o primeiro elemento;
- tipo = 2 => o pivô é selecionado pela mediana de três.

Após isso, existe uma função *ordena* cuida da criação das partições bem como de todas as chamadas recursivas realizadas para a criação de mais partições. Em uma implementação passada, existiam duas funções *ordena*, uma para os quicksort clássico, primeiro elemento e mediana de três e outra para o quicksort inserção. Contudo, devido à grande quantidade de código repetido a implementação foi reduzida para uma função que é capaz de realizar suas operações independentemente das entradas.

Já no método não recursivo não foi possível aproveitar do código já existente. Nesse caso foi utilizada uma lógica envolvendo o uso de pilhas, de forma a simular chamadas recursivas, as quais armazenam os vetores a serem ordenados posteriormente.

Felizmente todas as variações de quicksort aproveitam da mesma implementação da função *partição*, tendo como única diferença o modo de escolha do pivô. Abro um parêntese em relação a essa função devido ao fato de que as trocas de elementos ocorrem nela. Na minha opinião, ao trocarmos um elemento na posição X de um vetor com outro elemento que se encontra na posição Y, seguimos o seguinte algoritmo:

aux = elemento_1

elemento_1 = elemento_2

elemento_2 = elemento_1

Sendo assim, temos três movimentações ocorrendo quando consideramos a cópia de algum dos elementos para uma variável auxiliar. Contudo, seguindo os conselhos do monitor da disciplina, no trabalho foi considerado que o método de *swap* de elementos do vetor contaria

apenas como uma movimentação. Após gerar os gráficos considerando apenas uma movimentação, percebi que os resultados não estavam de acordo com minhas expectativas. Sendo assim, gerei os gráficos para três, movimentações e decidi considerá-los como objeto de análise nessa documentação. Contudo, todos os gráficos gerados considerando diferentes quantidades de movimentação estão anexados na pasta do projeto.

Por fim, de forma a contar a quantidade comparações e trocas de elementos, contadores foram sendo passados por referência e foram incrementados ao decorrer do programa, mais especificamente nas funções ordena, partição e inserção.

2.2. Entrada de Dados

Seguindo a documentação fornecida para a realização do trabalho, a entrada do programa de se estrutura da seguinte maneira:

./sort <variacao> <tipo> <tamanho> [-p]

- ./sort é o nome do executável gerado pelo makefile
- <variação>: tipo de quicksort a ser usado (os sete tipos diferentes citados na introdução)
- <tipo>: tipo do vetor de entrada (crescente – **OrdC**; decrescente – **OrdD**; aleatório - **Ale**)
- <tamanho>: tamanho do vetor (valor entre 50.000 e 500.000);
- [-p]: tag passado caso deseje imprimir os vetores utilizados no final do programa.

A obtenção desses argumentos é realizada de forma bem simples. Na linguagem C a função *main* recebe dois parâmetros por padrão: int argc e char *argv[]. O primeiro deles se refere a quantidade de argumentos que foram inseridos e o segundo armazena os valores propriamente ditos. Após a identificação deles, o programa se encarrega de gerar os vetores e realizar uma cópia para a impressão no final, caso o argumento [-p] seja passado.

2.3. Saída de Dados

Assim como a entrada de argumentos, a saída do programa também foi previamente definida no seguinte formato:

<variacao> <tipo> <tamanho> <n_comp> <n_mov> <tempo>

onde a *variacao, tipo* e *tamanho* são os parâmetros recebidos na entrada, *n_comp* e *n_mov* se referem ao número médio de comparações de elementos e movimentações efetuadas e *tempo* ao tempo mediano de execução (microssegundos)

3. Instruções de Compilação e Execução

Como já mencionado anteriormente, recomenda-se que a execução do código ocorra em um sistema Ubuntu versão 14 ou superior contendo pelo menos a versão 7 do GNU C Compiler (GCC). Foi utilizado um makefile que se encarrega de todas as operações de compilação dos arquivos ‘.c’.

Durante os testes realizados, percebeu-se que em vetores muito grandes erros de segmentação eram obtidos. Após pesquisas, o limite da pilha de execução do computador travado em 8 gigas era o culpado por tal situação. Sendo assim, é extremamente recomendado o aumento do tamanho da pilha antes da execução do trabalho. Para tal, basta executar o seguinte comando:

ulimit –s hard

4. Análise Experimental

Uma massiva quantidade de dados gerados foi considerada na análise que se segue. Foram realizados diversos testes com o intuito de checar a veracidade dos dados. Para definir a quantidade de testes a serem realizados, foi definida uma variável global (N_TESTES) a qual é responsável por determinar a quantidade de arrays a serem gerados. Durante os testes, foram feitos experimentos com N_TESTES de 5 a 20 com o intuito de obter valores bastante satisfatórios para o tempo. Por questões de organização, os gráficos e tabelas obtidos com os experimentos realizados estão armazenados no repositório do projeto no github (https://github.com/LucasPMM/Sorting_Algorithms) e na própria pasta do projeto. Além disso, os gráficos foram gerados com o auxílio do python, mais especificamente utilizando as bibliotecas numpy e matplotlib com jupyter notebook.

4.1. Análise de tempo

Uma primeira notação que podemos fazer em relação ao tempo seria ao tempo de execução do algoritmo. Devido a considerável quantidade de testes bem como os tamanhos dos vetores utilizados, o programa demora horas para completar sua execução como podemos ver na imagem 1:

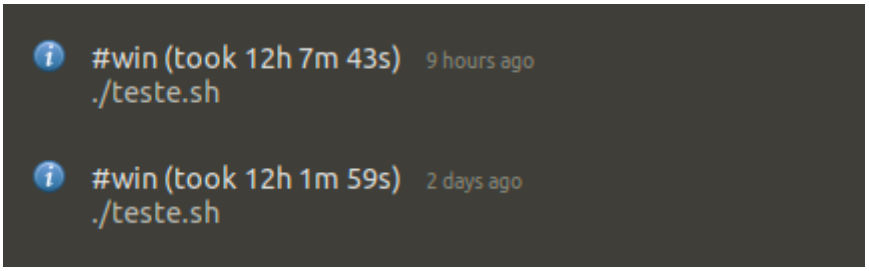

Imagem 1: Tempo de execução do programa.

Contudo, de forma a tentar diminuir esse tempo, foi adicionado o parâmetro -O3 após o gcc para que todas correções performáticas fossem corrigidas ou implementadas durante a compilação. Tal fato impactou na redução do tempo de execução como podemos ver abaixo:

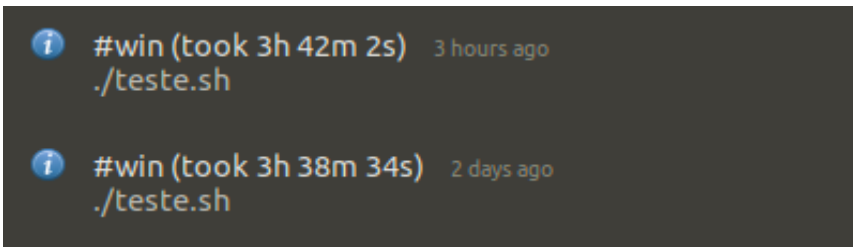

Imagem 2: Tempo de execução performático.

Sendo assim, recomenda-se que se mantenha tal implementação com o intuito de reduzir o tempo total gasto para o teste da aplicação.

Agora voltando a questão da análise de tempo, analisaremos os casos para vetores ordenados de forma crescente, decrescente e aleatórios, nessa respectiva ordem. Os gráficos, como mencionado anteriormente, podem ser encontrados tanto na pasta do projeto quanto no repositório do github.

Ao observamos o gráfico *OrdC_Tempo.png e OrdD_Tempo.png* é perceptível a gritante diferença entre a mediana do tempo de execução do QPE para os outros tipos de escolha de pivô. Durante os testes, cada execução chegou a durar cerca de 2 minutos cada. Esse fato ocorre, pois, as partições degeneradas são o resultado desse método de escolha de pivô, aumentando as chances de ocorrer o pior caso do quicksort – $O(n^2)$. Por outro lado, os quicksort do tipo inserção provaram ser bastante eficientes, sendo que o melhor deles é aquele que aplica o *insertionsort* nas 10% chaves finais do array. Isso ocorre devido ao fato de que um vetor ordenado entra no melhor caso da inserção – O(n).

O cenário muda bastante ao considerar vetores aleatórios (como podemos observar no gráfico *Ale_Tempo.png*). Os tipos de quicksort que apresentam os piores resultados são aqueles que implementam o método de ordenação por inserção, sendo o QI10 o mais demorado de todos. Pode-se explicar esse fato devido à ausência de ordenação dos vetores, o que implica no pior caso da inserção que é, por conseguinte, igual ao pior caso do quicksort e menor que o caso médio do mesmo. Por outro lado, as implementações do QM3 e QNR apresentam bons resultados.

4.2. Análise de comparações

Os gráficos de comparações são similares para os casos de vetores OrdD e Ordc (gráficos *OrdC_Comparacoes.png e OrdD_Comparacoes.png*). Como observamos na análise de tempo para os casos ordenados, o QPE demora mais que os demais e esse fato tem suas consequências nos gráficos de comparações. Podemos observar que o QPE lidera com folga no número de comparações seguindo uma função que apresenta um padrão exponencial. Os melhores casos estão novamente no quicksort inserção, pelos mesmos motivos já citados anteriormente. Mudando para o caso aleatório (gráfico *Ale_Comparacoes.png*) podemos observar que o uso do método de ordenação inserção pode cair no pior caso levando em consideração que o vetor não está ordenado. Sendo assim, sua implementação gera comparações excessivas.

4.2. Análise de movimentações

Os gráficos de movimentação foram aqueles que demonstraram as curvas mais atípicas. Para o caso ordenado em forma crescente (gráfico *OrdC_Movimentacoes.png*), podemos observar que o QPE realiza a maior quantidade de trocas devido ao fato que o pivô sempre troca de posição consigo. Já no caso ordenado decrescente (gráfico *OrdDMovimentacoes.png*), a aplicação não recursiva realiza o maior número de movimentações. Um fator interessante a ser analisado nesse caso é que o QC, QNR e o QM3 tem números de movimentações idênticos para todos os casos. Podemos explicar esse fator dado que o pivô é sempre o elemento central, fazendo com que as trocas sempre ocorram em cada elemento. Por fim, no caso aleatório (gráfico *Ale_Movimentacoes.png*), o método de inserção mais uma vez se encaixa no seu pior caso, realizando o maior número de movimentações quanto maior for a partição (1%, 5% ou 10%) dedicada a esse método de ordenação.

5. Conclusão

Com a realização de todos os testes e análise de dados, é notável que diferentes implementações têm por consequência resultados distintos. Os pontos mais notáveis seriam as implementações do quicksort com inserção que apresentam tempos satisfatórios em vetores ordenados e, em contrapartida, o quicksort do tipo QPE (pivô como primeiro elemento) apresenta o maior tempo de execução, uma vez que esse tipo de seleção do pivô implica em partições degeneradas.

Além disso, é necessário destacar que alguns erros de medida podem ter ocorrido devido a troca de contexto existente no processador do computador, fazendo com que processos percam o ganhem prioridade na execução e oscilando o tempo final do nosso programa. No entanto, enquanto os testes estavam sendo realizados, foi evitado o uso do computador de forma a não atrapalhar ou gerar qualquer travamento devido ao fato de executar vários programas ao mesmo tempo.

Sendo assim, a análise qualitativa de um algoritmo é de fundamental importância tendo que diversos sistemas reais muitas vezes não são capazes de suportar determinados 'piores casos' de execução de um algoritmo. Essa analise nos permite determinar qual seria a melhor escolha para determinado caso, pois nem sempre existem algoritmos melhores ou piores, mas há aqueles adequados a situações definidas.

6. Bibliografia

MEDIAN of three values. [*S. l.*], 15 fev. 2018. Disponível em: https://stackoverflow.com/questions/48799800/median-of-three-values. Acesso em: 5 jun. 2019.

QUICKSORT. [*S. l.*], 13 maio 2019. Disponível em: https://pt.wikipedia.org/wiki/Quicksort. Acesso em: 5 jun. 2019.

FEOFILOFF, Paulo. Quicksort. São Paulo, 3 out. 2018. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/quick.html. Acesso em: 5 jun. 2019.

Chaimowicz, L. e Prates, R. "Ordenação: Quicksort", pdf disponível no Moodle da turma de Estruturas de Dados 2019/1 da UFMG. Acesso em: 5 jun 2019.

Autor Desconhecido. "Quicksort". Disponível em: https://www.geeksforgeeks.org/quick-sort/. Acesso em 5 jun. 2019.