Trabalho Prático II: O Retorno

Lucas S. Teles

1. Introdução

Esta documentação descreve a modelagem e implementação dos algoritmos de ordenação necessários para o segundo Trabalho Prático da disciplina de Estruturas de Dados, semestre 2020/2 da UFMG.

Os algoritmos foram implementados em uma classe chamada Sort, que possui vários métodos de ordenação e de utilidades. Os testes foram feitos com uma medição interna utilizando ferramentas da linguagem C++, a metodologia de testes será detalhada na seção adequada.

Na seção 2, a implementação dos algoritmos será descrita com mais detalhes; na seção 3, instruções de compilação e execução serão apresentadas; na seção 4, será feita a análise de complexidade de tempo e espaço do sistema completo; na seção 5 se encontram gráficos e tabelas que facilitam a comparação dos métodos e uma descrição mais detalhada do ShellSort; na seção 6 será detalhada a metodologia de testes; na seção 7 serão feitos alguns comentários a respeito das dificuldades encontradas; finalmente, na seção 8, estará a conclusão da documentação com comentários sobre o percurso durante a solução do problema.

2. Implementação

Nenhuma estrutura de dados foi necessária já que a linguagem fornece funcionalidade padrão para vetores de inteiros e de “strings”. A escolha de não implementar uma estrutura de dados, foi tomada visando reduzir a complexidade do sistema, já que o escopo do trabalho é a implementação dos algoritmos e a análise deles.

Primeiramente a classe Sort era um namespace como uma sugestão encontrada na internet indicava. Porém, após uma revisão a implementação via classe se mostrou mais chamativa.

Os algoritmos de ordenação foram implementados baseados nos códigos mostrados em aula e apenas pequenas modificações foram necessárias (a grande maioria relacionadas ao limite de ordenação de um subvetor, especialmente para evitar Segmentation Fault).

Existem alguns métodos utilitários que são chamados pelos de ordenação para ajudar em sua execução, todos eles são privados para encapsular o funcionamento interno da classe permitir ao usuário utilizar apenas os algoritmos padrão, sem necessidade de manipulação de nenhum dado além do nome do arquivo e da quantidade de linhas a serem lidas.

Os arquivos seguem algum tipo de padronização na ordem dos #include (primeiro os arquivos próprios, depois as bibliotecas da linguagem), na declaração de métodos e variáveis, e implementação de métodos. As declarações e a implementação dos métodos é dividade em seções que são similares de alguma forma, além de serem indentadas a fim de facilitar a legibilidade do código. Na declaração os métodos são separados entre os privados e públicos, já na implementação eles estão, de certa forma, agrupados de maneira a agilizar a leitura de algum outro método

chamado pelo método que o leitor está acompanhando. Por quase todo o sistema existem comentários, alguns separando as seções similares outros para melhor entendimento do código.

Alguns tratamentos de erros são feitos, se supõe que todos os casos problemáticos foram cobertos.

O Sistema Operacional utilizado para testar o programa foi o Windows 10, a linguagem utilizada foi C++, o compilador foi o Mingw-w64. O processador é um Intel Core i3-2310M e o computador possui 6 GB de RAM.

3. Instruções de Compilação e Execução

Para compilar o programa, basta executar o comando make no terminal no diretório TP2 (onde se encontra o arquivo Makefile), sem necessidade de nenhum argumento adicional à chamada do comando. Os arquivos objeto .o vão aparecer no diretório obj e o executável run.out (como pedido na descrição e no Makefile disponibilizado) será compilado no diretório bin.

O programa requer dois argumentos adicionais ao ser chamado, a localização e nome do arquivo dos planetas e a quantidade de linhas do arquivo a serem lidas, nessa ordem, como descrito na especificação do Trabalho.

Os arquivos DEVEM respeitar a formatação apresentada na proposta do Trabalho para funcionalidade total e correta do sistema. Porém, mensagens de erro indicarão ao usuário o que impediu o funcionamento do programa.

Nenhum tratamento foi feito para o segundo argumento com um número de linhas maior do que a quantidade de linhas presentes no arquivo. Porém acredita-se que tal situação não incorrerá em erros.

4. Análise dos Algoritmos

Todos os algoritmos implementados possuem complexidade de espaço O(n) (já que o MergeSort não foi implementado).

O método de Inserção é estável, adaptável muito simples de implementar e excelente para arquivos pequeno ou ordenados. O pior caso é $O(n^2)$ para movimentações e comparações. O melhor caso é O(n).

O HeapSort não é estável, não é adaptável (pois se deve criar o Heap e ordená-lo toda vez que o algoritmo é executado, independente se o vetor já está ordenado, também por essa razão não é indicado para vetores pequenos). Tem pior e melhor casos O(n * log n). Indicado para situações onde se deseja a todo custo evitar um pior caso mais lento.

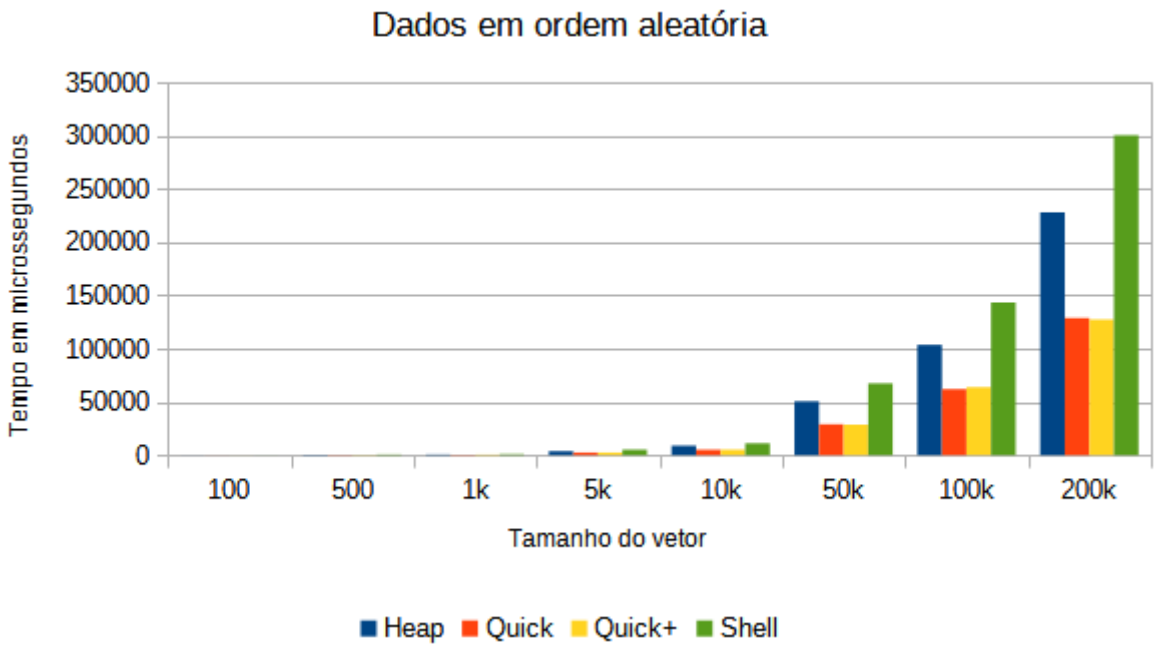
O QuickSort, como o nome já diz, é o algoritmo mais eficiente de que se tem conhecimento. Porém ele não é estável, mas é adaptável. Tem pior caso $O(n^2)$, porém é fácil diminuir a chance dessa situação acontecer, por isso mesmo tem caso médio O(n * log n), que, com exceção de alguma constante, é idêntico ao seu melhor caso.

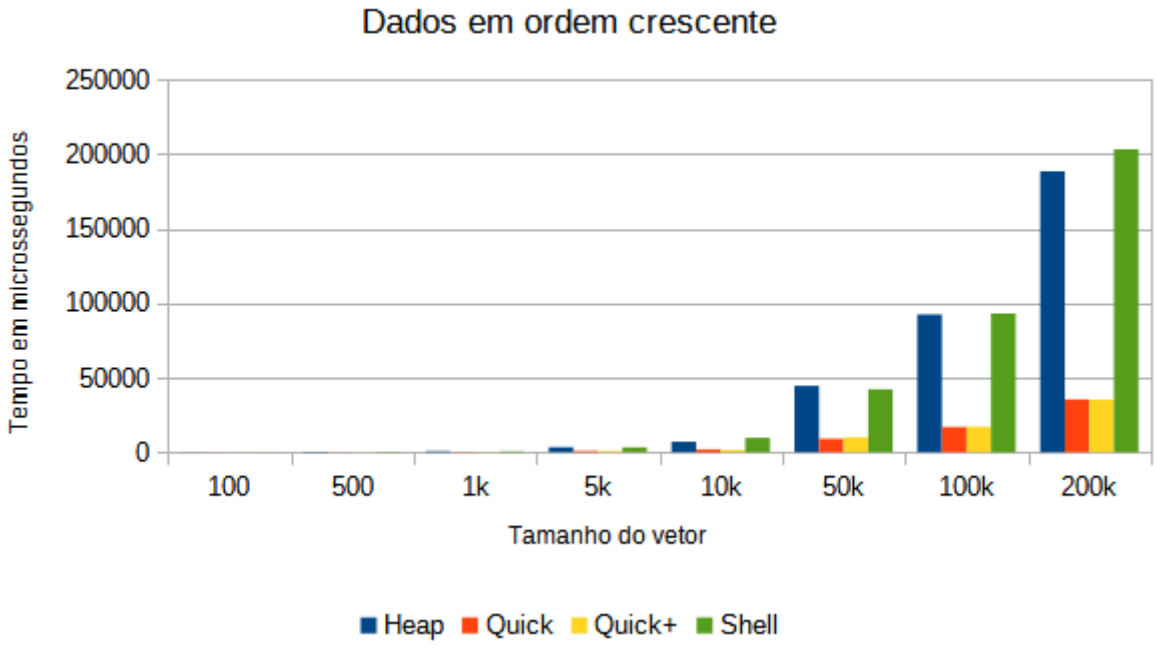
O QuickSort é mais rápido que o HeapSort por uma constante nos casos médio e melhor e no geral é o mais indicado, feito a ressalva que sua implementação é um tanto mais complexa que os outros algoritmos.

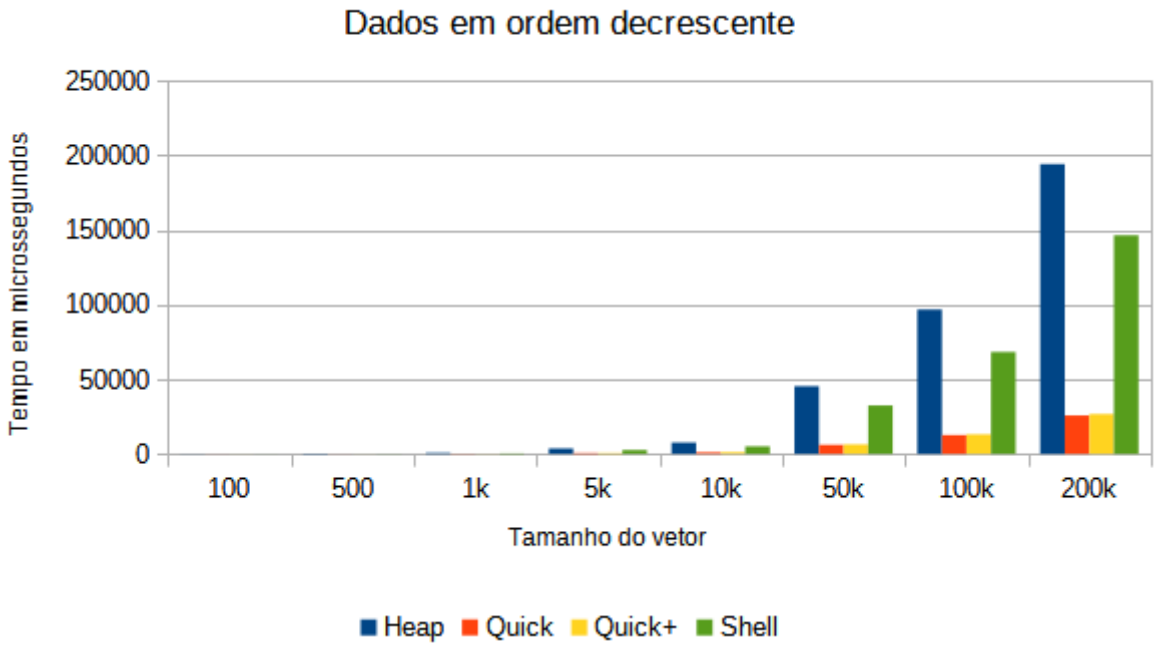
O método QuickSort+ (como será chamada a versão melhorada do QuickSort nesse documento) possui todas as características do QuickSort, entretanto, como discutido anteriormente, o primeiro visa evitar o pior caso do QuickSort. Se utilizados em situações cotidianas com vetores em ordem aleatória, ao longo do tempo o QuickSort+ se mostra mais eficaz, porém a diferença não é tão grande quanto escolher entre um outro algoritmo e o QuickSort.

O método ShellSort será discutido com mais detalhes em uma seção posterior, porém, vale dar um breve descrição. Ele é uma melhora no método da Inserção que permite dar “passos” maiores do que uma posição no vetor para comparar e movimentar os valores.

5. Gráficos e Tabelas







O método da Inserção foi ocultado dos gráficos para permitir uma melhor visualização.

Se vê pela escala e proporção da barra em azul que o HeapSort não é adaptável. Se faz claro também o quanto o QuickSort é, de fato, rápido.

O ShellSort ordena o vetor baseado no funcionamento do método Inserção, porém, ele o percorre várias vezes, com saltos que começam grandes (como metade do tamanho do vetor) e vão diminuindo a cada iteração até que eles tenham tamanho 1 (o mesmo tamanho dos saltos do método Inserção). Quando os saltos têm tamanho 1 o método percorre o vetor já ordenado como o Inserção faria e como o último é linear para um vetor ordenado, esse passo é, de certa forma, instantâneo.

Em outras palavras, o método da Inserção é um caso especial do ShellSort no qual os saltos têm tamanho 1 e apenas uma passagem pelo vetor.

O ShellSort é bastante simples de se implementar e apresenta ganhos incríveis em relação ao método da Inserção. Como se pode ver no gráfico, ele tem tempo de execução semelhante ao HeapSort, sendo até mais rápido em algumas situações.

A eficiência do método está diretamente ligada aos incrementos escolhidos para os saltos. Várias sequências de incrementos foram propostas (teoricamente) e descobertas (empiricamente). Muito do que diz respeito à análise de complexidade do método é desconhecido ou apenas mostrado empiricamente, por essa razão não existe uma resposta apenas para a complexidade assintótica de cada caso. Entretanto, o pior pior caso conhecido é $O(n^2)$. melhor pior caso é $O(n * \log^2 n)$, os melhores casos mais comuns são $O(n * \log n)$, até por isso método se assemelha tanto ao HeapSort. O pior melhor caso conhecido é também $O(n * \log^2 n)$.

A escolha do método ShellSort foi feita baseada em dois fatos importantes. O primeiro é que o ShellSort é um método bastante simples porém com uma análise extremamente complexa e que ainda possui problemas em aberto. O segundo se trata do fato de que sua implementação pode ser feita utilizando o método Inserção com apenas duas mudanças. Um loop externo à chamada do

método de inserção que determina a sequência de incrementos e modificar o Inserção para “dar os saltos” de acordo com os valores da sequência. Dessa forma, o mesmo método da Inserção pode ser usado para os dois casos, para o Inserção padrão basta chamar o método uma vez para o vetor todo, começando da segunda posição, enquanto que para o ShellSort são feitas várias chamadas começando da posição que tem o valor do incremento.

No código se encontram 3 sequências de incrementos para a execução do ShellSort. A primeira delas é a proposta pelo criador do método, que começa com metade do tamanho do vetor e vai sempre dividindo por 2. Essa sequência está presente apenas por referência, já que ela não é tão eficiente, pois os valores que se encontram em posições ímpares (que seriam as pares na linguagem C++, já que os vetores começam da posição 0) só serão ordenados na última execução do método, a que possui salto de tamanho 1.

A segunda foi obtida por pura sorte porém inspirada em uma sequência encontrada no livro de Robert Sedgewick. A terceira é a sequência mais eficiente conhecida, descoberta por Marcin Ciura, que foi “hardcoded” no programa para demonstrar três possíveis formas de usar as sequências para executar o método.

A sequência de Ciura vai até 1750, os valores restantes foram calculados usando um fator de 2,25.

A seguir uma tabela com os valores do tempo de execução do método da Inserção.

Linhas Ordem	100	500	1k	5k	10k	50k	100k	200k
Aleatório	999µs	3ms	7ms	216ms	869ms	22s 586ms	89s 383ms	355s 229ms
Crescente	1ms	4ms	21ms	497ms	1s 763ms	43s 366ms	174s 436ms	725s 873ms
Decres.	1ms	1s	999µs	998µs	997µs	2ms	5ms	13ms

Existe uma inconsistência no uso das línguas Portuguesa e Inglesa no código fonte. Alguma liberdade foi tomada para escolher os melhores termos (de acordo com o autor) para cada variável e função a fim de descrever melhor o propósito delas.

6. Metodologia de Testes

Com exceção do método de Inserção, todos os testes executaram os métodos 100 vezes para cada arquivo para cada quantidade de linhas. Uma média aritmética do tempo de execução foi calculada.

O método de Inserção foi executado somente uma vez para cada arquivo para cada quantidade de linhas.

O cálculo do tempo foi feito com o auxílio da biblioteca <chrono> e apenas a execução da chamada à função dos algoritmos é medida. Dessa forma, operações irrelevantes para a análise, como abertura e leitura do arquivo, entre outras, são ignoradas.

7. Dificuldades

A intenção inicial era implementar um IntroSort (ou como melhoria do QuickSort ou como o algoritmo extra), porém isso se mostrou mais difícil do que pensado inicialmente e não foi possível fazê-lo em tempo hábil.

Algumas outras dificuldades menores serão discutidas na sessão seguinte.

8. Conclusão

Muitas dificuldades pequenas surgiram, especialmente em relação aos limites dos subvetores entre outras, como os valores passados para as funções de utilidade. Até por isso que a conclusão desse trabalho requereu uma boa capacidade de depuração do autor e é parte do que de melhor se experimentou no percurso de feitura.

Foi necessária bastante criatividade para, não só tornar o método de Inserção adaptável (no sentido de poder ser usado também pelo ShellSort), mas também visualizar os valores corretos para a utilização do ShellSort e do Inserção padrão. Essa experiência foi, talvez, a mais marcante.

A revisão da modelagem feita, permitiu uma melhor organização do código e o resultado serve para demonstrar o valor de um código legível e organizado.

Bibliografia

Chaimowicz, L. e Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Sedgewick, R. (1998) *Algorithms in C.* 3rd ed. Addison-Wesley Publishing Company, Inc.

https://stackoverflow.com/questions/2808398/easily-measure-elapsed-time

https://en.cppreference.com/w/cpp/chrono/duration

https://yourbasic.org/golang/quicksort-optimizations/

https://en.wikipedia.org/wiki/Shellsort

https://oeis.org/A102549

https://www.cs.princeton.edu/~rs/shell/paperF.pdf

https://www.geeksforgeeks.org/shellsort/