Documentação - Trabalho Prático 2 – Estruturas de Dados

Mateus Latrova Stephanin - 2019006981

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

mateusls@ufmg.br

1. Introdução

Esta documentação lida com o problema de ordenar uma lista de duzentas mil bases instaladas em diversos planetas em ordem decrescente de acordo com a distância a que se encontram da Terra.

Para solucioná-lo, foram implementados alguns algoritmos de ordenação bem conhecidos, alguns mais eficientes(Heap Sort e Quick Sort), de complexidade linearítmica na média, e outros menos(Insertion Sort e Shell Sort), de complexidade, no máximo, quadrática.

Aqui, tem-se o objetivo de explicar o motivo das decisões tomadas ao longo da elaboração do trabalho e esclarecer a ideia de alguns dos algoritmos incluindo aspectos gerais de sua implementação. Ademais, far-se-á uma análise com um pouco mais de profundidade a respeito da complexidade de cada algoritmo individualmente, tendo como referência os dados coletados, além de comparar os diferentes casos para cada algoritmo e, também, verificar as semelhanças e diferenças de algoritmos distintos para o mesmo caso de entrada.

2. Instruções de compilação e execução

A implementação estará no diretório “projeto”, que conterá os diretórios “bin”, “include”, “obj” e “src”. Acesse o diretório “projeto” pelo terminal e, nele, execute o comando “make” para a compilação e ligação do código. Após a compilação, o arquivo executável “run.out” estará no diretório “bin”. Acesse-o pelo terminal e o execute passando como parâmetro, respectivamente, o *path* do arquivo que contém a lista de bases e o número N de linhas do arquivo a serem ordenadas pelo programa. Por exemplo, se o *path* do arquivo é “../dados.txt” e o número N é 500, ficaria da maneira a seguir: “./run.out ../dados.txt 500”.

3. Análise dos algoritmos

Insertion Sort:

Na elaboração do código desse algoritmo, foi usada a seguinte ideia: no começo, considera-se que o array de n elementos possui apenas o primeiro elemento. Depois, serão feitas n-1 iterações, sendo que na iteração de número i($1 <= i < n-1$), o elemento na posição

i do *array* será inserido nessa lista inicial já na posição correta. Ao final, todos os elementos terão sido inseridos na posição correta e, portanto, o *array* estará ordenado.

A complexidade desse algoritmo é quadrática no pior caso e no caso médio, pois para cada um dos n-1 números, são feitas no máximo m comparações a fim de inseri-lo na lista, sendo m o número de elementos já presentes nela(1<= m <= n-1). Porém, no melhor caso, isto é, quando o vetor já está ordenado, sua complexidade é linear, já que serão feitas apenas n-1 comparações para todas as inserções.

Além disso, ele é estável, já que a inserção garante que a ordem relativa entre elementos de mesma chave não é mudada, e, também, adaptável, tendo em vista que seu comportamento muda dependendo da entrada, como vimos acima na análise de sua complexidade.

Heap Sort:

Primeiramente, em relação à sua implementação, transforma-se o vetor dado na entrada num min-heap, visto que a ordem desejada é decrescente, utilizando o método rebuildHeap() num loop(esta parte equivale ao método Constroi() visto em aula). Depois, faz-se outro loop em que se troca o primeiro elemento do array, que é o mínimo, com o elemento na posição i(i variando de n-1 a 1) e reconstrói-se o min-heap, para fazer o mesmo processo até que i seja igual a n-1.

Já no aspecto de complexidade assintótica, o algoritmo possui custo de ordem linearítmica em todos os casos possíveis, o que o caracteriza como inadaptável, visto que seu comportamento e sua performance são sempre muito próximos independentemente da ordem presente no vetor antes de ser ordenado. Por fim, pode-se afirmar que ele não é estável devido às trocas feitas para a construção e reconstrução do min-heap.

Quick Sort:

Aqui, a implementação usada foi bem similar à utilizada nas aulas. Primeiro, faz-se a partição, utilizando o elemento central do vetor como pivô, deixando elementos maiores que ele à sua esquerda e menores à sua direita. Por fim, chama-se recursivamente a função para as duas partes divididas, o que fará com que esse processo se repita até que o vetor esteja ordenado de forma decrescente.

Em relação à sua complexidade, ele possui custo de ordem $O(n \cdot \log n)$ no melhor caso e, também, no caso médio. Porém, no pior caso, ou seja, quando o vetor está ordenado de forma decrescente e o pivô escolhido é sempre o último element, pode ter ordem quadrática. Isso mostra que, dependendo da configuração do vetor de entrada, o algoritmo muda seu comportamento, o que o classifica como adaptável. Por fim, visto que o processo de partição pode mudar a ordem relativa de elementos de mesma chave, o algoritmo é não-estável.

Quick Sort melhorado:

A melhoria escolhida para o algoritmo Quick Sort foi a mediana de 3, pois ela é uma mudança simples no código que gera uma transformação significativa. Ela faz ele ser comparável aos algoritmos Heap Sort e Merge Sort em termos de adaptabilidade, pois evita a ocorrência do pior caso, o qual trazia, antes, um custo de ordem quadrática.

A estratégia utilizada na implementação desse processo foi a seguinte: seleciona-se os elementos que estão na primeira posição, na posição central e na última posição. Faz-se a ordenação em ordem decrescente desses elementos no próprio vetor, ou seja, o maior fica na primeira posição, o mediano fica na posição central e o menor, na última. Depois, troca-se o elemento do meio com o penúltimo. Usamos esse novo penúltimo elemento como pivô da partição, sendo que os elementos no início e no final do vetor não precisam participar dela, já que o procedimento da mediana garante que, nesse momento, eles estão na partição correta. A partir daí, a implementação segue exatamente igual à do método sem a melhoria.

Shell Sort:

Esse algoritmo foi escolhido pela sua inteligente estratégia, originalidade e, também, pois, dentre os pesquisados, foi um dos poucos que não utiliza os algoritmos vistos em aula para resolver parte da ordenação. A ideia dele é analisar o vetor com intervalos(em inglês, *gaps*), isto é, comparar os elementos do vetor com outros que distam de certo intervalo e, caso o da esquerda seja menor que o da direita, efetua-se uma troca. Esse intervalo começa sendo metade do tamanho do vetor, diminuindo pela metade até que chegue a ser igual a 1, quando acontece uma situação curiosa: o algoritmo se comporta como o Insertion Sort. Entretanto, o vetor já está quase ordenado, favorecendo a performance do algoritmo de ordenação por inserção(quase um melhor caso do insertion).

A implementação dessa ideia foi feita com alguns loops aninhados de forma a realizar a comparação da maneira explicada acima. Para tal, utilizaram-se dois contadores: um que era incrementado de um e outro que era decrementado do tamanho do intervalo. Ao fim do loop externo, divide-se o intervalo por dois e repete-se o processo até que ele atinja o valor zero.

Em relação à sua complexidade, não se pode afirmar nada. Até hoje, ninguém foi capaz de provar qual é a exata complexidade assintótica desse algoritmo. Todavia, há duas conjecturas a respeito dela: uma em que o número de comparações realizadas encontrado é da ordem de n elevado a 1,25 e outra em que é $O(n \cdot (\log n)^2)$.

Algumas de suas vantagens são sua implementação enxuta e o bom desempenho em arquivos de tamanho moderado. Enquanto que suas desvantagens consistem na ausência de estabilidade e na sua adaptabilidade, isto é, a dependência causada pela ordem da entrada no seu tempo de execução, característica nada interessante para a grande maioria das aplicações.

4. Comparações entre o resultado dos algoritmos

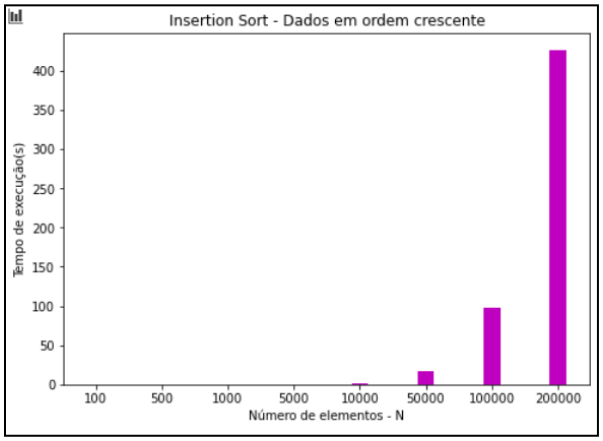

Figura 1

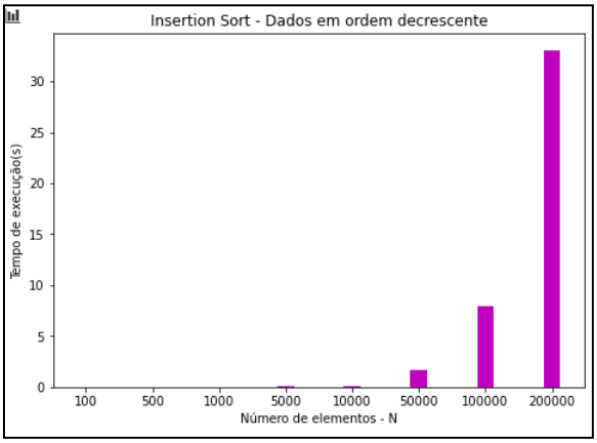

Figura 2

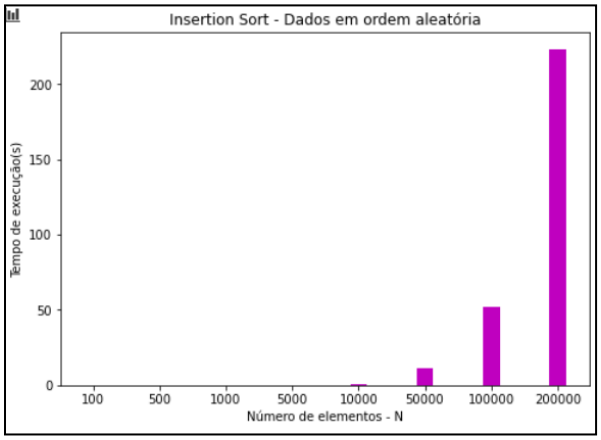

Figura 3

Como esperado, devido à sua complexidade quadrática, o algoritmo de ordenação por inserção foi o mais lento de todos. Consequentemente, foi separado do gráfico dos demais por conta de questões de ordem de grandeza e escala. Conforme se pode ver acima, para a maior quantidade de linhas do arquivo(duzentas mil), o tempo mínimo gasto para execução, dentre os três casos, foi de aproximadamente 32s(no caso da ordem decrescente), enquanto que o tempo máximo gasto pelos outros algoritmos em todos os casos foi de 0.45s, como será apresentado mais à frente.

Porém, como se pode observar no gráfico da figura 4, há um caso em que o algoritmo é bem mais rápido que nos demais casos para qualquer número de linhas: o caso em que o arquivo já está ordenado.

Essa situação já era prevista e é exatamente o melhor caso de complexidade para esse algoritmo, no qual ele possui um custo apenas linear.

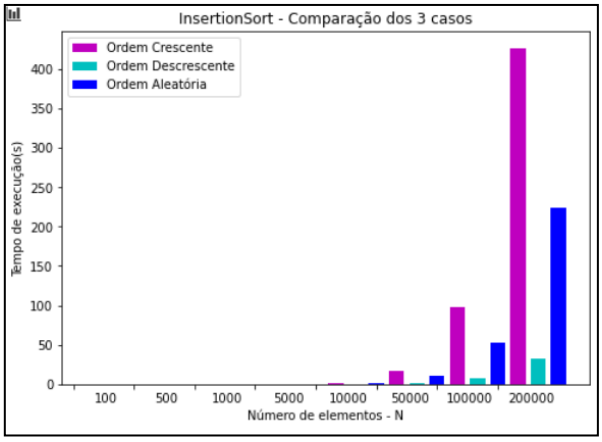

Figura 4

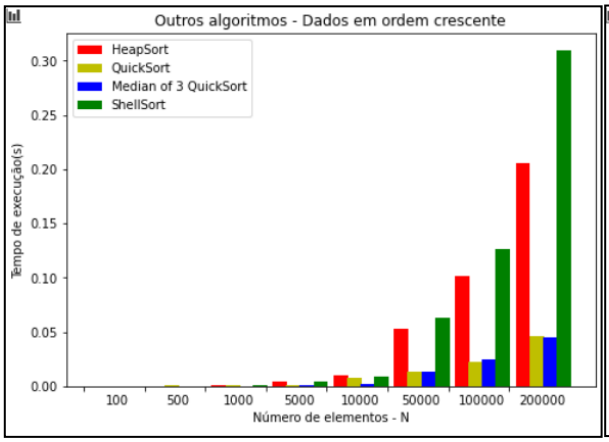

Figura 5

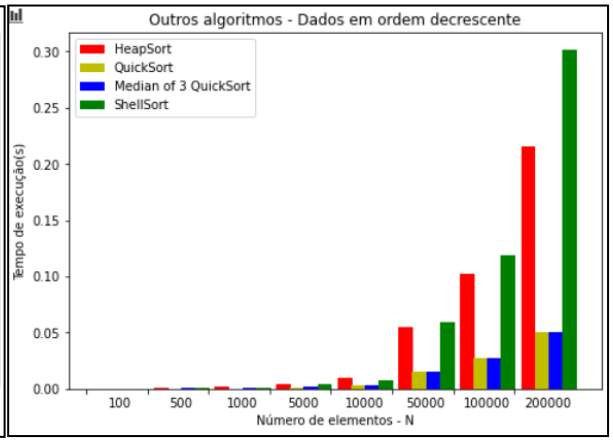

Figura 6

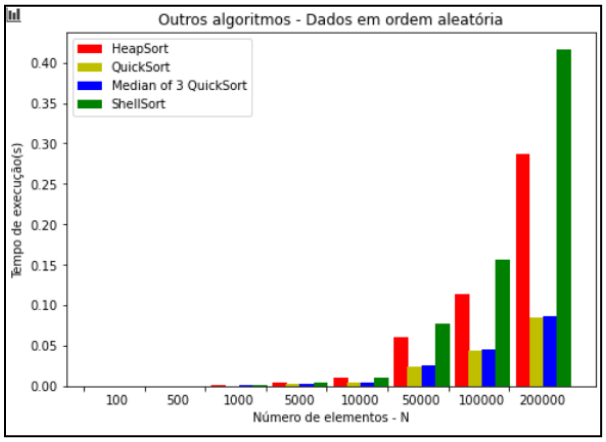

Figura 7

Agora, partindo para os algoritmos restantes, há algumas observações e comparações relevantes a serem feitas. Primeiramente, percebe-se que o algoritmo mais rápido de todos é o Quick Sort, tanto na versão comum quanto na versão melhorada, já que seus tempos de execução são muito próximos em todos os casos de entrada. Aqui, sua versão melhorada não

fez uma diferença significativa no tempo gasto devido ao fato de que não houve a ocorrência do pior caso. Caso contrário, a melhora seria facilmente observada nos resultados.

Além disso, um ponto importante é a visível constância do método Heap Sort devido à sua inadaptabilidade. De acordo com as figuras 5,6 e 7, pode-se verificar que a variação que o seu tempo de execução sofreu é desprezível.

Por fim, deve-se ressaltar a eficiência do algoritmo Shell Sort. Mesmo que ele seja o mais lento quando o número de linhas é maior ou igual a cinquenta mil, para valores menores do que esse, ele tem um tempo semelhante ou, até mesmo, menor que os algoritmos bem conhecidos por sua grande eficiência, Heap Sort e Quick Sort. Isso lhe traz uma importância e visibilidade diferenciada na lista gigante de algoritmos de ordenação conhecidos.

5. Conclusão

Este trabalho lidou com o problema da ordenação das bases para a volta da missão "Extração-Z" e resolveu-o com uma abordagem clara, organizada e com a maior simplicidade alcançada pelo autor. O paradigma de programação utilizado foi o de orientação a objetos, criando-se duas classes para representar as entidades do problema. Procurou-se fazê-lo de forma que qualquer programador pudesse entender a solução apenas analisando o código, utilizando nomes mnemônicos, indentação adequada e comentários em pontos relevantes do código.

Durante a implementação da solução e a coleta e análise dos dados de tempo(coletados com o auxílio da bibioteca <chrono> da STL) houve diversas dificuldades, as quais se tornaram grandes ganhos em conhecimento para a realização de futuros projetos. Foi possível praticar muitos conceitos antes vistos, porém que não haviam ficado claros sem uma implementação mais robusta e uma comparação detalhada.

Dentre os aprendizados obtidos mediante as dificuldades, podem ser listados:

- Medição de tempo de execução de qualquer parte de um programa;
- Construção de gráficos usando a biblioteca matplotlib da linguagem Python;
- Melhor entendimento de como a ordem de complexidade de um algoritmo influencia na prática;
- Classes e funções *friend.*

6. Bibliografia

- https://www.youtube.com/watch?v=9crZRd8GPWM
- https://www.youtube.com/watch?v=ddeLSDsYVp8&t=277s
- https://www.youtube.com/watch?v=1Vl2TB7DoAM&t=224s
- https://www.techiedelight.com/measure-elapsed-time-program-chrono-library/
- http://www.cplusplus.com/
- https://stackoverflow.com/
- https://www.geeksforgeeks.org/