Documentação de projeto: Retornando para Casa

Augusto Carvalho Porto Pereira

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brazil

1.Introdução

Após resolver o problema de explorar uma base extraterrestre utilizando robôs semi-autômatos instruídos remotamente, foi necessário descobrir como garantir que todas essas máquinas retornassem à Terra com segurança. Para isso, verificou-se que a melhor solução para garantir um retorno realmente efetivo é trazer os robôs de volta começando do planeta mais distante até o planeta mais próximo.

Procurando definir qual é a melhor maneira de ordenar esses planetas e dar uma resposta para um problema tão importante para a humanidade foi necessário implementar um sistema que realize testes em vários algoritmos de ordenação. A partir dos relatórios de execução do sistema será decidido qual o algoritmo utilizado na missão.

Na seção 2 está descrito um breve passo-a-passo para que o sistema possa ser executado. Na parte 3 estão comentários sobre cada um dos algoritmos utilizados na implementação além de detalhes específicos, seguindo para a seção 4 onde estes algoritmos serão comparados em termos de tempo de execução. Na quinta e última seção está uma conclusão sobre as dificuldades e aprendizados do trabalho.

2. Instruções de compilação e execução

Para executar o programa, o usuário deverá criar um arquivo de tipo texto (.txt) com os dados de entrada. Cada linha deverá conter o nome do planeta e a distância numérica referente aos saltos hiperespaciais da Terra até esse planeta. O valor da distância deverá ser um número inteiro e ele deverá ser separado do nome do planeta por um caractere de espaço. Preferencialmente, esse arquivo deve ser criado dentro da pasta “TP2”.

Criados os arquivos a serem avaliados, o usuário deve abrir o terminal, acessar a pasta “TP2” e executar o comando “bin/run.out [nome do arquivo].txt [número de linhas]”. O valor do campo “numeroLinhas” é referente ao número de linhas do arquivo que serão lidas e ordenadas. As linhas seguintes não serão analisadas naquela execução. Para definir qual algoritmo de ordenação será utilizado, é necessário alterar o valor da constante global *ALGORITMO_ORDENACAO* com um valor inteiro entre 1 e 5 referente aos algoritmos, respectivamente: Insertion Sort, Heap Sort, Quick Sort, Quick Sort Modificado e Cocktail Sort. Por padrão, o algoritmo Insertion Sort está pré-selecionado.

Ao executar o comando descrito acima, será impresso na saída padrão (‘stdout’) a lista dos 7 (ou menos) planetas mais distantes da Terra, a partir dos quais o retorno para casa iniciará. Nenhum arquivo de entrada será alterado durante a execução do programa.

O sistema foi testado utilizando um Windows Subsystem for Linux (WSL) que instancia a versão 18.04 do Ubuntu. O código foi totalmente implementado em C++ (além, é claro, do arquivo Makefile) utilizando o GCC C++ Compiler (g++). A execução dos testes foi realizada em uma máquina com 8GB de memória RAM e um processador i3 de 3.30Ghz.

3. Implementação

Para desenvolver a solução foi escrito um programa relativamente simples, centrado principalmente em um único arquivo principal onde está implementada a leitura do arquivo contendo os dados de entrada além da descrição e chamada dos 5 métodos de ordenação utilizados na solução. Além desse arquivo, a classe *Planeta* foi criada para auxiliar na interpretação dos dados da leitura, contendo os campos *nome* e *distancia* (da Terra). Para além da classe Planeta, foi criada a classe Pilha utilizando tipos genéricos em c++ para ser utilizada especificamente na implementação do Quick Sort Modificado, que será descrito mais abaixo.

O primeiro algoritmo de ordenação implementado como padrão na execução do sistema é o **Insertion Sort**. onde a lista de elementos é lida da esquerda para direita e a cada novo item lido ele é deslocado para a esquerda para que fique em ordem crescente. O método foi implementado de maneira **estável** e adaptável, pois no **melhor caso**, onde a lista já está ordenada, sua complexidade é **O(n)**, enquanto no **pior caso**, no qual a lista está em ordem decrescente, sua complexidade é **$O(n^2)$.**

O segundo algoritmo desenvolvido foi o **Heap Sort**. Nele, é construída uma árvore de prioridades, chamada de *heap,* comparando os nós pais e os nós filhos alterando a posição deles para que o maior elemento vá para a primeira posição da árvore em toda construção. A ordenação é realizada ao trocar de lugar o primeiro elemento do vetor, que é o maior, com o último elemento, para em seguida reconstruir a árvore com todos os elementos que não são o maior e realizando essa ação sucessivamente para que todos os elementos estejam ordenados. É um método **não estável** porém independente da entrada, sua complexidade é sempre **O(n log n).**

O terceiro método de ordenação implementada é o **Quick Sort**.um dos métodos mais famosos e eficazes conhecidos. Seu processo consiste em subdividir a lista em partições menores, que se divide entre elementos maiores e menores que um pivô selecionado (elemento no meio do vetor), e repetir esse método para ordenar todas as partições.O método **não é estável**. Quando o pivô é obtido selecionando sistematicamente o maior ou menor elemento do vetor, configurando o **pior caso**, a complexidade é **$O(n^2)$.** Já para o melhor caso e para o caso médio, onde simplesmente o pivô não é escolhido da forma descrita anteriormente, a

complexidade é **O(n log n)**. Em geral, essa é a complexidade considerada ao se comparar o Quick Sort com outros algoritmos.

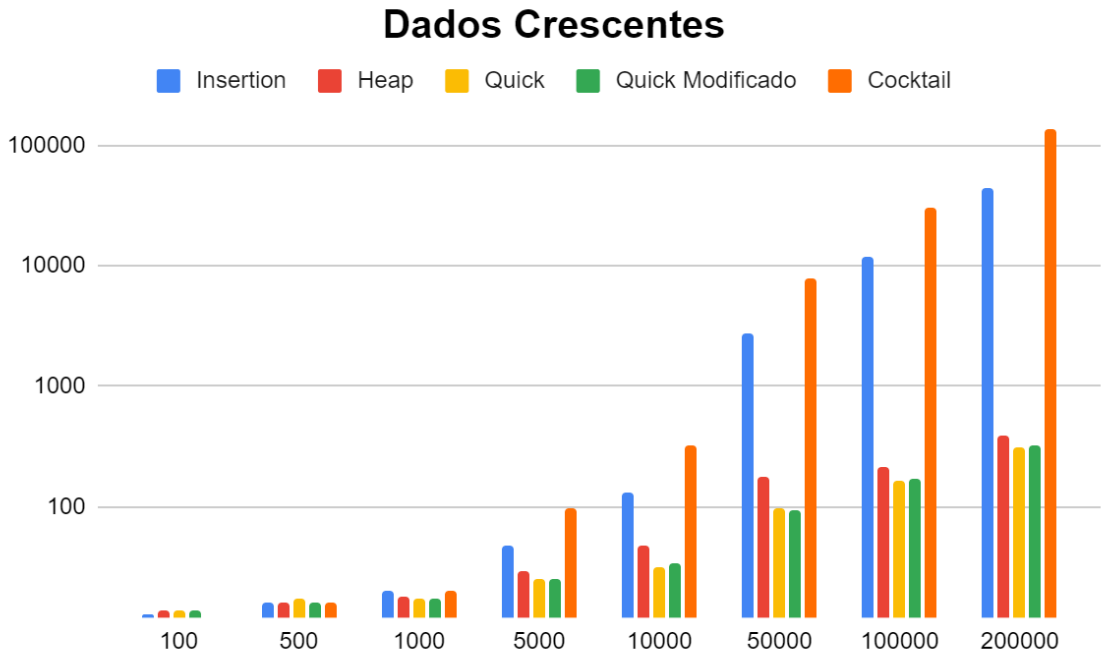
A quarta implementação é uma modificação do algoritmo anterior, o qual recebeu o nome de **Quick Sort Modificado.** A mudança realizada consiste em implementar uma pilha responsável por administrar as partições ao invés de gerenciar um algoritmo recursivo. Essa mudança pode auxiliar a leitura e interpretação do funcionamento do algoritmo, pois não é trivial compreender a execução de um código recursivo. Embora seja moroso, visto que é necessário criar uma pilha para implementá-lo, é interessante mostrar a possibilidade de conversão de um método recursivo para um não recursivo. Esse algoritmo compartilha a **não-estabilidade** e a complexidade média **O(n log n)** do Quick Sort comum.

Por fim, o quinto e último algoritmo escolhido foi o **Cocktail Sort**, também chamado de Bubble Sort Bidirecional. Sua execução é bem similar ao bubble sort, porém a cada varredura do vetor o elemento movimentado será diferente: quando o deslocamento é do início ao fim do array, o algoritmo movimenta para o fim o maior elemento. Em seguida, se deslocando do fim ao início do array, o método irá mover, um a um, o menor elemento para o início do vetor. Ele é chamado de Cocktail Sort pois os maiores e menores elementos irão se ordenando pouco a pouco e se "fechando" como em uma coqueteleira de drinks. Esse método foi escolhido pois sua visualização é bem didática, além de ser mostrado como uma solução para o problema dos coelhos e tartarugas. O método **não é estável** e sua complexidade para o **pior caso** e **caso médio** são **O(n^2),** mas tendendo a O(n) caso a lista esteja parcialmente ordenada**.** O melhor caso é sempre **O(n)**, no qual a lista já está corretamente ordenada.

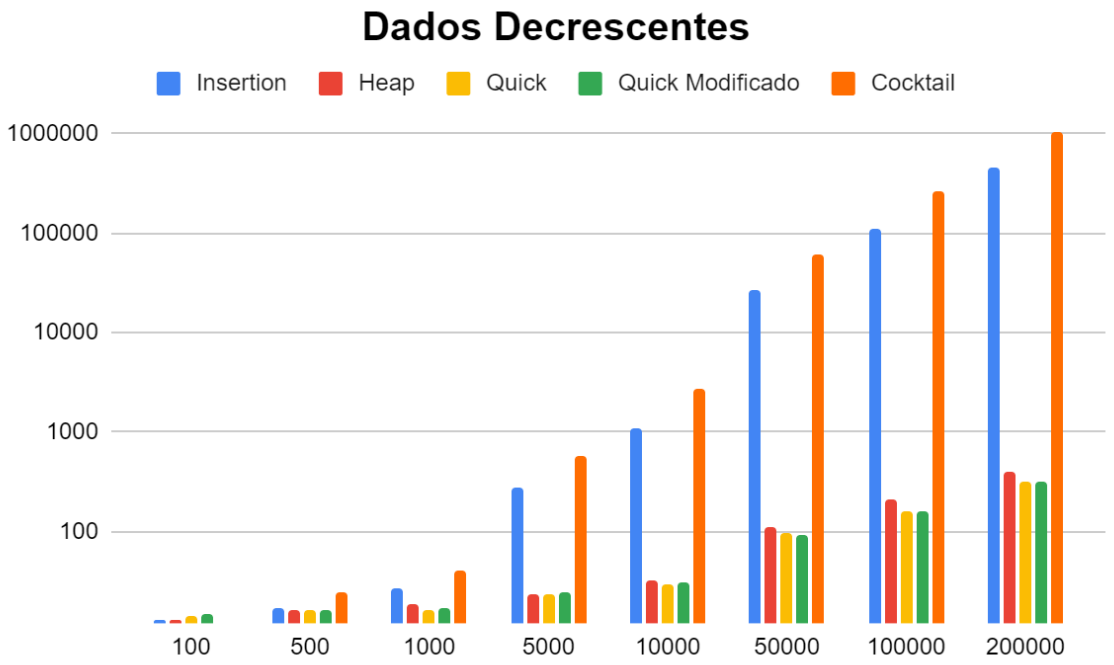
4. Comparações

Para que o objetivo do trabalho possa ser concluído, além de implementar cada um dos algoritmos de ordenação é necessário realizar as comparações entre eles e identificar qual a melhor escolha para auxiliar a humanidade nessa importante missão. Para essas comparações, foi testado o desempenho de cada método de ordenação para três tipos de entradas diferentes e, em cada uma, verificando o tempo gasto pelo sistema para realizar os cálculos para a leitura de 100, 500, 1.000, 5.000, 10.000, 50.000, 100.000 e 200.000 planetas por execução. Os resultados foram calculados em milissegundos.

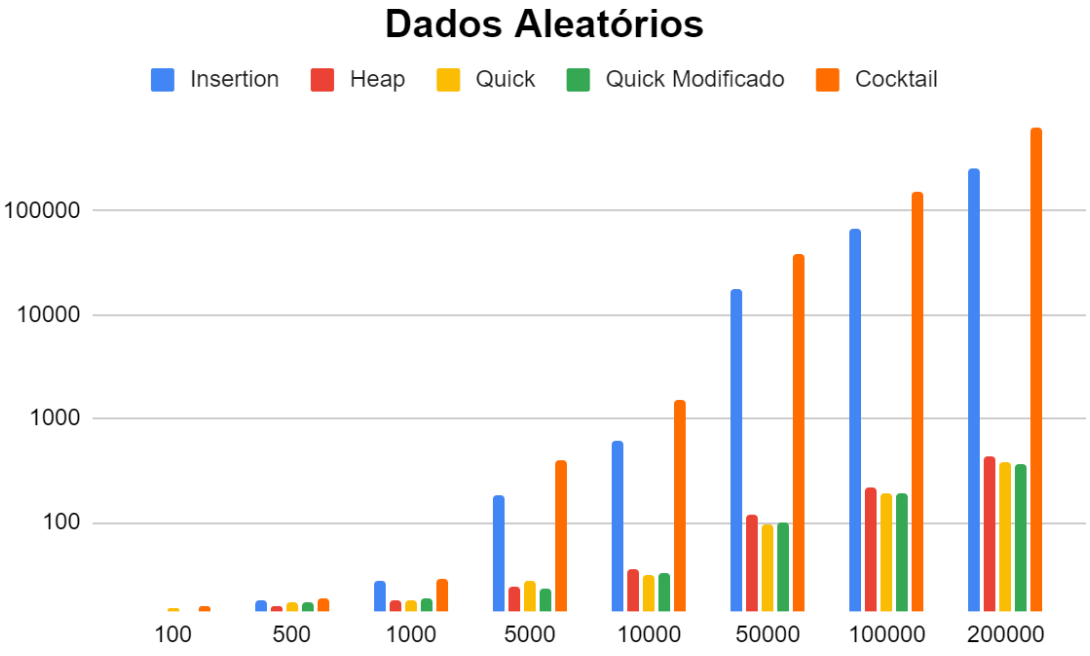
Primeiramente, vamos verificar no gráfico abaixo o que acontece quando o programa recebe uma lista de planetas já ordenada em ordem crescente.Podemos ver um grande equilíbrio entre *Heap* e *Quick* (e sua modificação), enquanto há uma dominância grande do *Cocktail* seguido do *Insertion,* mesmo que na teoria estes sejam os melhores cenários para estes algoritmos.



Analisando o desempenho em para entradas pré-ordenadas em ordem decrescente, os desempenho do *Heap* e dos *Quick's* se mantiveram praticamente inalterados, embora o tempo gasto pelo *Insertion* cresça ainda mais, visto que este é seu pior caso, e o Cocktail chega a atingir a casa de 1.000.000 milissegundos, ou aproximadamente 17 minutos.



Por fim, foram realizados testes para entradas em ordem aleatória, sem nenhuma organização específica. Essa é uma boa comparação a se realizar para comprovar a eficácia de um algoritmo de ordenação pois é possível chegar cada vez mais perto do caso médio de cada método. Como pode ser verificado no gráfico abaixo, as execuções mostram que o *Insertion* e o *Cocktail* não são métodos interessantes para resolver o problema, considerando o seu desempenho ruim para entradas muito numerosas.



Entre as curiosidades advindas da análise do gráfico, foi possível comprovar que o desempenho do *Heap* praticamente não se altera para qualquer tipo de entrada, como foi afirmado em teoria.Além disso, considerando o *Cocktail* uma variação do algoritmo *Bubble*, é possível verificar que o algoritmo *Insertion* é razoavelmente superior mesmo dentre os algoritmos simples, mesmo que não se compare ao *Heap* ou *Quick*.

Por fim, graças ao esforço de análise dos dados, conseguimos concluir que o algoritmo que aparenta ser a solução ideal para o nosso problema é o *Quick Sort*, pois seu desempenho tende a ser sempre melhor ou igual do que o desempenho dos outros algoritmos. A decisão, porém, de utilizar ele em sua versão recursiva ou em sua versão não-recursiva implementada por pilhas pode ser escolhida pelo próprio time de desenvolvimento, pois como o desempenho das duas versões é praticamente similar, o ideal é que a equipe escolhe a opção com a qual estiver mais confortável.

5. Conclusão

Esse trabalho foi focado não somente no desenvolvimento de um sistema envolvendo diversos algoritmos de ordenação, mas também em levantar dados e comparar execuções, fomentar o aprendizado de análise de performance do sistema e exibir gráficos comparativos. Esse processo, mesmo que não seja sobre escrever linhas de código, é muito importante no área da tecnologia da informação.

A experiência prática de implementar algoritmos de ordenação já visitados durante as aulas e buscar novos algoritmos ou mudanças nos algoritmos conhecidos é um bom exercício de fixação do conhecimento técnico, além é claro de reforçar os aprendizados de análise de complexidade assintótica vistos anteriormente.

Referências

Chaimowicz, L. and 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.

Gráficos - Google Sheets.

HackerEarth, Sorting Algorithms
(https://www.hackerearth.com/practice/algorithms/sorting)