UFC

UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS-RUSSAS

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PAULO HENRIQUE DINIZ DE LIMA ALENCAR

ESTRUTURA DE DADOS AVANÇADAS – TRABALHO 1 - PARTE 1

RUSSAS

2021

1 VISÃO GERAL

Primeiro trabalho na disciplina de estrutura de dados Avançadas. O trabalho consiste na implementação dos algoritmos: MaxHeap, Heap-Sort, Insertion-Sort e Tabela-Hash. Após a implementação, os algoritmos deveriam ser testados com diferentes instâncias, contextos e métodos.

2 ESPECIFICAÇÕES GERAIS

- Linguagem de implementação: C;
- Máquina utilizada na análise dos algoritmos: notebook ACER-Aspire ES15 ES1-572-3562, SSD de 100 Gb, 11 Gb de memória RAM (DDR4) e processador Intel Core i3-6006U (2.0 GHz L3 Cache);
- Sistema Operacional: Linux – Zorin-OS – 64 bits;
- Compilador: gcc versão 9.3.0.

3 ALGORITMOS DE ORDENAÇÃO: HEAP-SORT E INSERTION-SORT

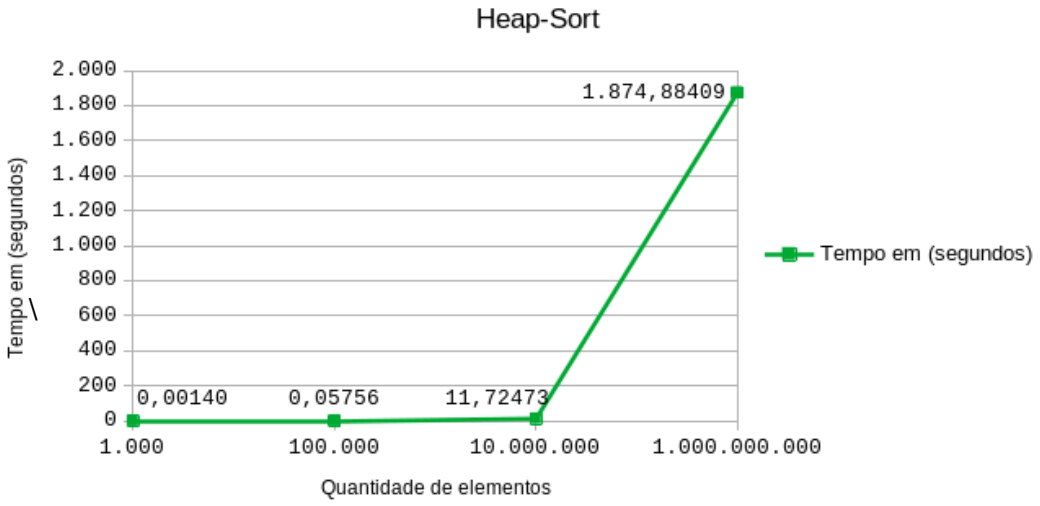
3.1 Análise Empírica

Seguindo os critérios estabelecidos pelo professor da disciplina, foram gerados 4 arquivos com valores pseudoaleatórios (inteiros) com tamanhos: 1.000, 100.000, 10.000.000 e 1.000.000.000. Cada arquivo foi lido e seus valores foram transferidos para um determinado vetor. Em seguida, o vetor foi ordenado pelo Heap-Sort e Insertion-Sort para cada um dos tamanhos citados acima. O tempo gasto por cada algoritmo na ordenação, baseado no comprimento do vetor serão apresentados graficamente nos tópicos seguintes.

> **Observação:** os arquivos com números pseudoaleatórios foram gerados utilizando um *script* python através da biblioteca *numpy* (A biblioteca numpy possui uma função random que garante uma boa aleatoriedade). Além disso, os números pseudoaleatórios inseridos nos arquivos possuem um range (0, 2.000.000) em média.

3.2 Resultados - Heap-Sort

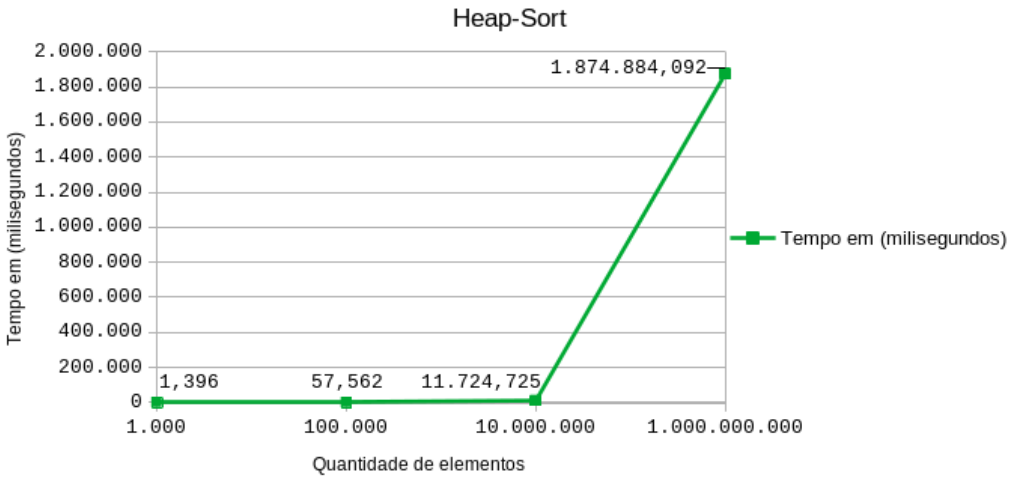
Gráfico 1 - Tempo gasto em segundos pelo heapSort para realizar a ordenação de *mil*, *cem mil*, *dez milhões* e *um bilhão de elementos* aleatórios presentes em um vetor.



Fonte: elaborado pelo autor.

Com base no gráfico, notamos que o HeapSort demorou em média 1.874 segundos (31 minutos) para ordenar 1 Bilhão de valores aleatórios dentro de um range de [0, 2.000.000].

Gráfico 2 - Tempo gasto em milissegundos pelo heapsort para realizar a ordenação de *mil*, *cem mil, dez milhões* e *um bilhão* de elementos aleatórios presentes em um vetor.



Fonte: elaborado pelo autor.

3.2.2 Insertion-Sort

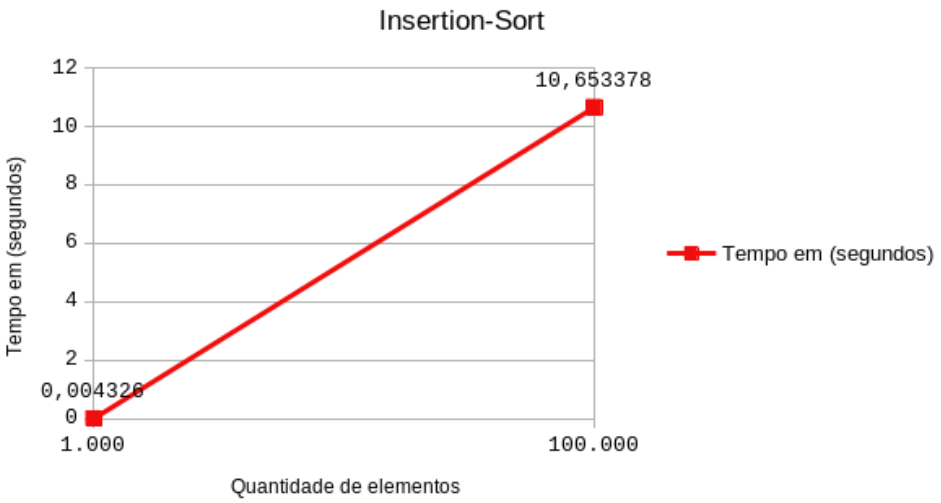
Em relação a análise empírica dos dados solicitadas pelo professor fazendo uso do Insertion-Sort, foram obtidos os seguintes resultados:

1. Foram conseguidos somente o tempo de ordenação para 1.000 elementos e 100.000 elementos;

2. O algoritmo Insertion-Sort tentou ordenar durante 24 horas os valores com entrada 10.000.000. Após esse período, foi decidido encerrar o processo e desligar a máquina;

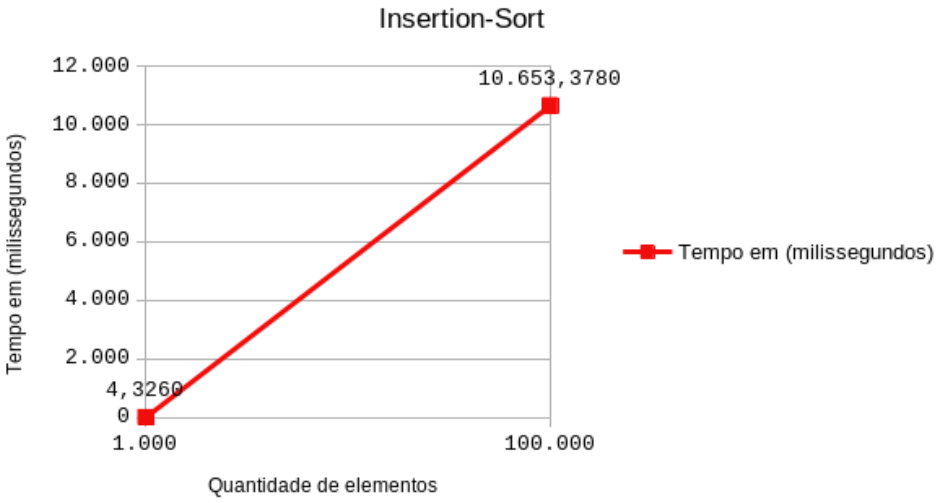
3. Ainda foi pensado na possibilidade de obter um tempo máximo no pior caso (vetor ordenado decrescentemente). O tempo máximo poderia ser obtido fazendo alguns experimentos para encontrar o tempo médio de execução de uma operação no computador e depois multiplicando esse tempo por n^2 (complexidade do Insertion-Sort).

Gráfico 3 - Tempo gasto em segundos pelo Insertion-Sort para realizar a ordenação de *mil* e *cem mil* elementos aleatórios.



Fonte: elaborado pelo autor.

Gráfico 4 - Tempo gasto em milissegundos pelo Insertion-Sort para realizar a ordenação de *mil* e *cem mil* elementos aleatórios.



Fonte: elaborado pelo autor

4. COMPARANDO HEAP-SORT E INSERTION-SORT

Comparando o Heap-Sort e o Insertion-Sort somente com os valores que foram possíveis realizar os testes, obtemos as seguintes observações.

No teste com entrada de 1.000 valores o Heap-Sort realizou a ordenação em 0,001396 segundos. Já o Insertion-Sort levou 0,004326 segundos. Portanto, com a entrada de *mil* valores o Heap-Sort foi aproximadamente 3x mais rápido que o Insertion-Sort.

Já com a entrada de 100.000 valores, o Heap-Sort realizou a ordenação em 0,057562 segundos. Por outro lado, com a mesma entrada o Insertion-Sort levou 10,653378 segundos, cerca de *185x* mais lento que o Heap-Sort.

Para as entradas de 10 milhões e 1 Bilhão não foi possível realizar a contagem de tempo de execução para a ordenação fazendo uso do Insertion-Sort. Assim, não foi possível realizar a comparação com esses valores para os dois algoritmos.

Essas comparações mostram que uma complexidade de $\Theta(n * log\ n)$ (complexidade do Heap-Sort) realmente faz grande diferença quando comparado com algum algoritmo de complexidade $\Theta(n^2)$(complexidade do Insertion-Sort).

5. COMPARAÇÃO DOS MÉTODOS DE DISPERSÃO DA TABELA HASH

Como citado nas especificações deste trabalho foram necessário criar tabelas com tamanhos de m = 100.000.

- 1 tabela para o Método da divisão;
- 1 tabela para o Método da multiplicação;
- 1 tabela para o Método da dobra.

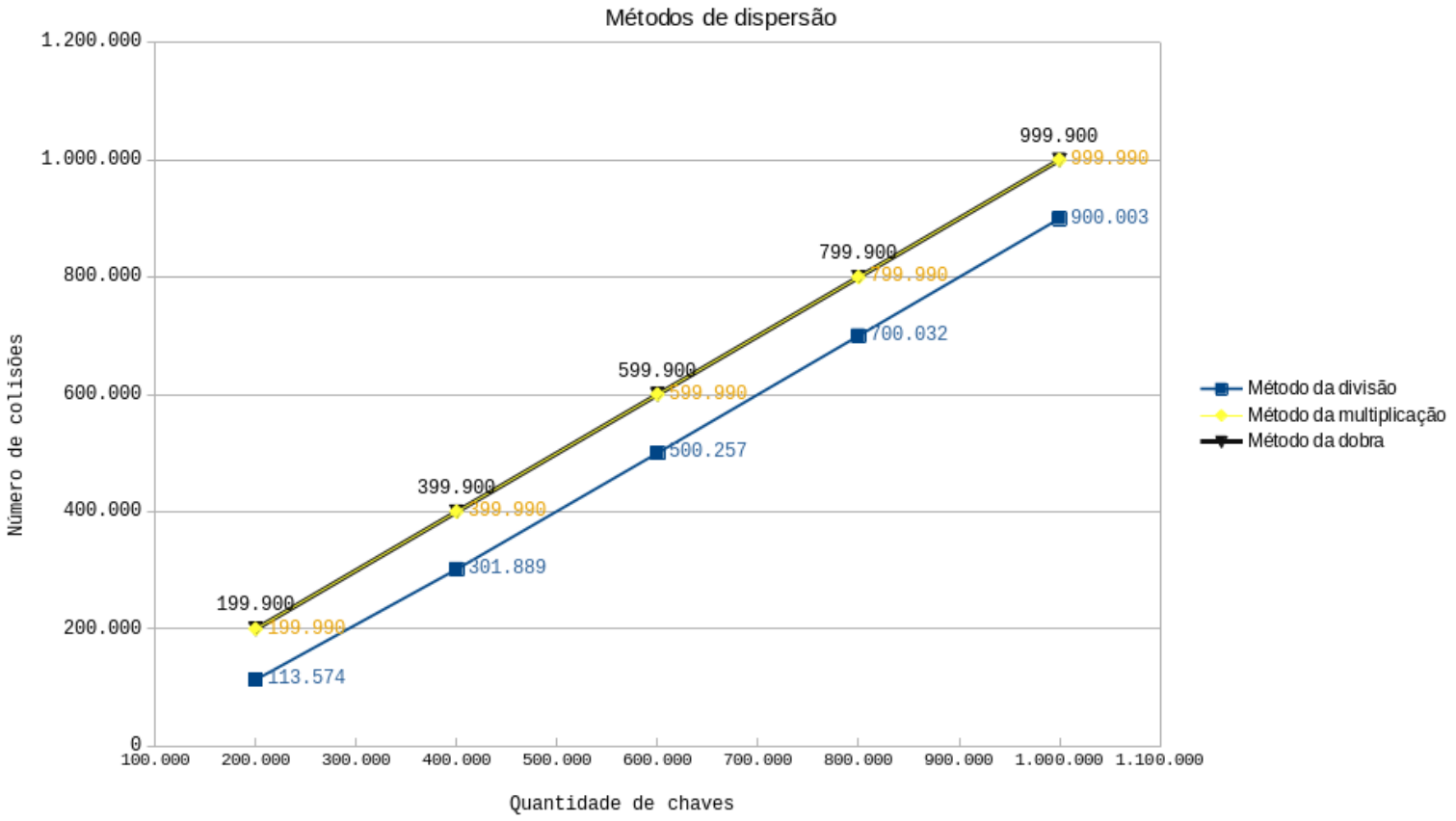
Em seguida realizei os testes de colisões com vetores de chaves com o n's:

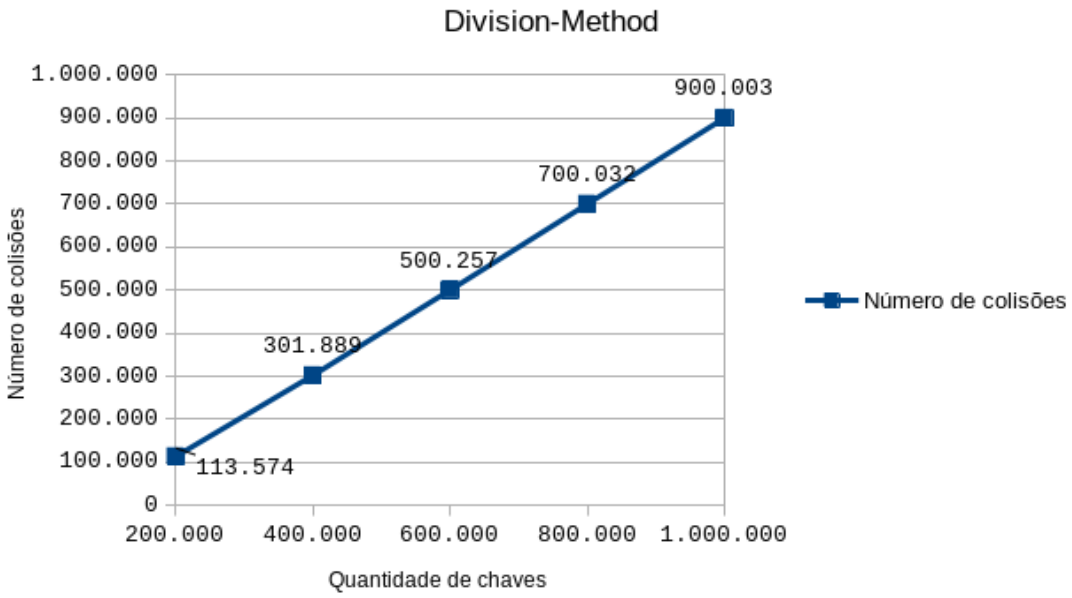
n
200.000
400.000
600.000
800.000
1.000.000

Fonte: elaborado pelo autor

Os números aleatórios colocados nos vetores de chaves, foram inseridos em arquivos .txt com um range de [0, 2.000.000.000] em seguida colocados nos vetores de chaves para suas respectivas operações.

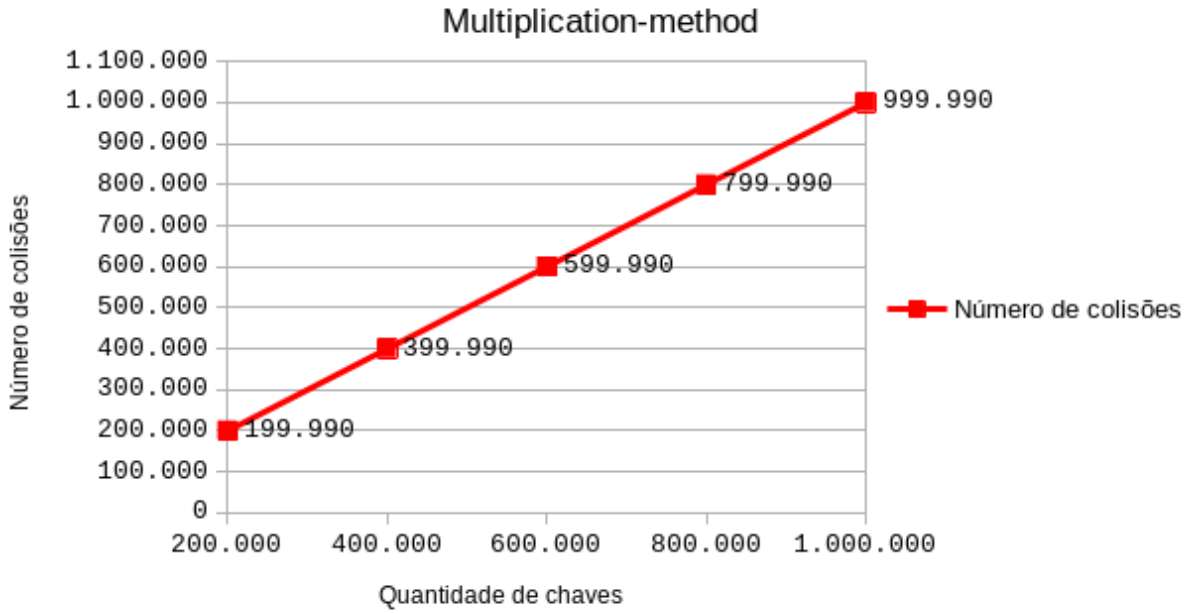
Gráfico 5: representa (quantidade de chaves n) × (nº de colisões) em que é mostrado as 3 curvas (uma curva para método da divisão, multiplicação e dobra) .





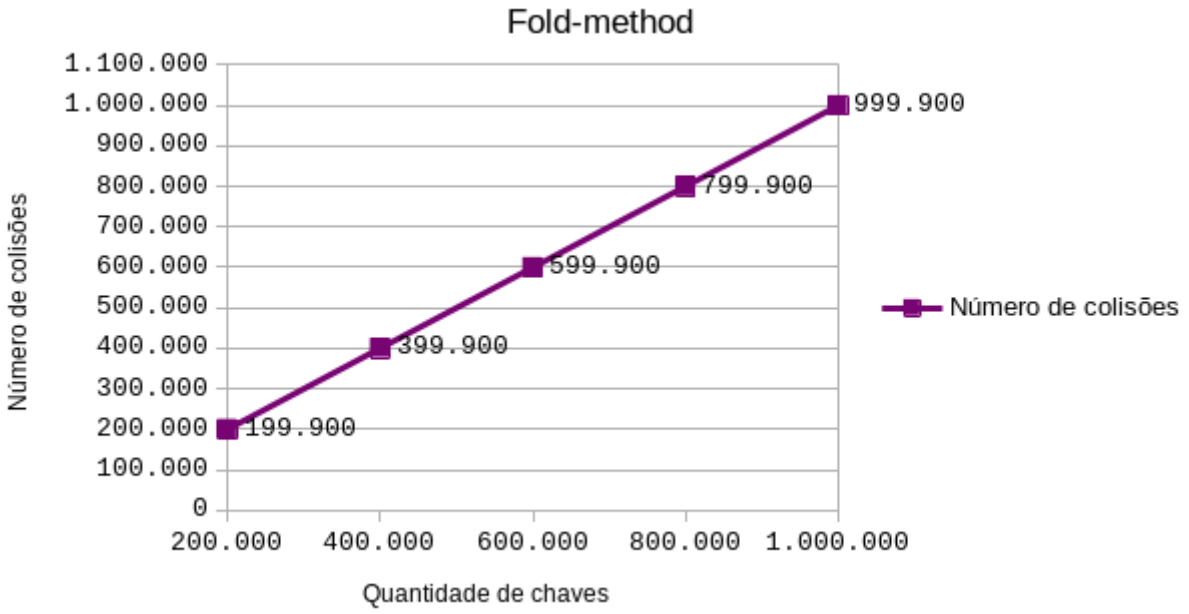
Fonte: elaborado pelo autor

Gráfico 6: análise empírica com o método da multiplicação.



Fonte: elaborado pelo autor

Gráfico 9: análise empírica com o método da dobra.



Fonte: elaborado pelo autor

Assim, com base nessas análises e entradas (m = 100.000, n = 200.000, n = 400.000…) podemos concluir que o método da divisão foi o que gerou um menor número de colisões.

5. ARQUIVOS *.TXT* GERADOS:

Para a realização dos testes, foi decidido gerar arquivos com extensão *.txt* onde foram inseridos os números pseudoaleatórios. No entanto, é importante salientar que isso não alterou o tempo de execução dos algoritmos, pois todos os arquivos foram gerados com antecedência.

Além disso, durante a produção dos algoritmos não foram contabilizados os tempo levado para pegar os números presentes nos arquivos e inserir nos vetores. No caso dos algoritmos de ordenação foram contabilizados somente o tempo que a função demorava para ordenar determinada entrada.

Outro ponto importante, foi que, a função ***random*** presente na biblioteca ***numpy*** do python foi responsável por gerar os números aleatórios. Por que foi feito isso? Essa biblioteca é escrita boa parte com a linguagem C possuindo uma grande capacidade de gerar números pseudoaleatórios bastante distintos. Após gerar esses arquivos, que foram utilizados nos testes, cada número presente no *.txt* foi inserido em determinado vetor.