Bruno Carvalho

Diogo Destefano

Pedro Bellotti

Rafael Terra

**Estrutura de Dados II**

Relatório Referente à Parte 1:

- Algoritmos de Ordenação

- Palavras do Momento

- Hashing

**UNIVERSIDADE FEDERAL DE JUIZ DE FORA**

25 de Setembro de 2017

**Índice**

1. Introdução......................................................................................................... 03
2. BubbleSort........................................................................................................ 04
3. Dados Brutos.................................................................................................... 05
4. Analise dos Resultados..................................................................................... 21
5. Dados Sobre o Desenvolvimento..................................................................... 23

**1 – Introdução**

Desenvolvido em linguagem C++ e utilizando a IDE Visual Basic, a primeira parte do projeto tem como objetivo analisar e comparar os diferentes algoritmos de ordenação e funções de Hashing, a fim de se concluir quais métodos se apresentam mais eficientes em determinados contextos.

Variáveis como número de entradas, gasto em memória e tempo de execução foram determinantes para definir e comparar os resultados dentre os algoritmos testados.

O projeto foi desenvolvido visando consumir uma quantidade reduzida de memória, visto que por vezes, os algoritmos foram testados com 1.000.000 entradas. A cada iteração, o conjunto de dados foi *randomizado* a fim de se obter um resultado mais confiável (gasto de randomização: O(n)).

Já na segunda parte, foi utilizada a mesma linguagem e IDE da primeira parte. Neste projeto, os textos dos Tweet`s lidos da base de dados foram analisados e o número de palavras com tamanho maior ou igual a dois caracteres foram contabilizados utilizando funções de hash, para assim, obter-se a frequência de cada palavra contida nos textos.

**2 – BubbleSort**

O BubbleSort foi o algoritmo escolhido como o algoritmo não visto em sala, devido a sua popularidade e a sua facilidade para implementação, sendo um dos algoritmos de ordenação mais simples.

Uma forma de trabalhar com o algoritmo BubbleSort é comparando os elementos adjacentes (dois a dois), por exemplo: compara-se a primeira posição do vetor com a segunda, na segunda iteração (repetição), compara-se a segunda posição do vetor com a terceira, e assim sucessivamente. De acordo com o algoritmo, podemos ordenar o vetor de forma crescente ou decrescente.

O algoritmo BubbleSort percorre todo o vetor diversas vezes. No melhor caso, o algoritmo executa **n** operações relevantes, onde **n** representa o número de elementos do vetor. No pior caso, são feitas **n²** operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, não é recomendado o uso dele para aplicações que requerem velocidade ou trabalhem com uma grande quantidade de dados.

Essa ordenação lembra como as bolhas num tanque de água que procuram seu próprio nível, e disso vem o nome do algoritmo, Bubble Sort.

**3 – Dados Brutos**

-Análise dos QuickSorts

-Tabelas

As tabelas a seguir foram criadas usando os dados retornados por cada tipo de QuickSort com N variando entre 1000, 5000, 10000, 50000, 100000, 500000 e 1000000.











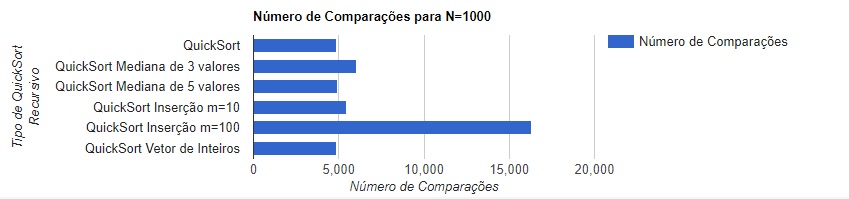


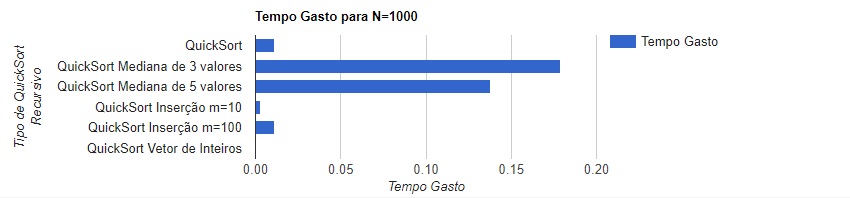


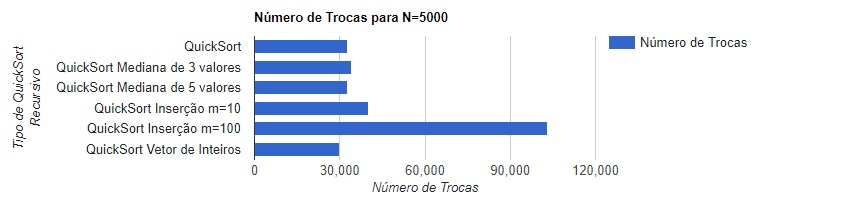
-Gráficos

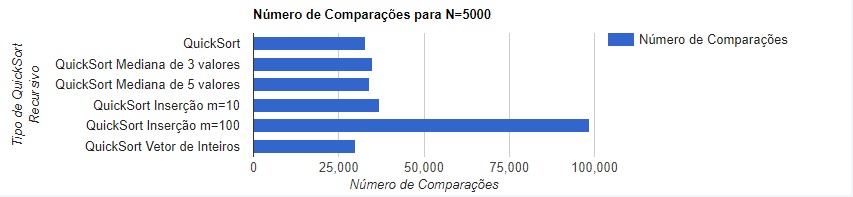
Os gráficos a seguir foram criados usando os dados retornados por cada tipo de QuickSort com N variando entre 1000, 5000, 10000, 50000, 100000, 500000 e 1000000.

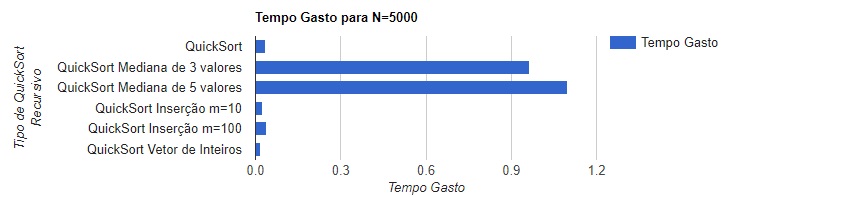


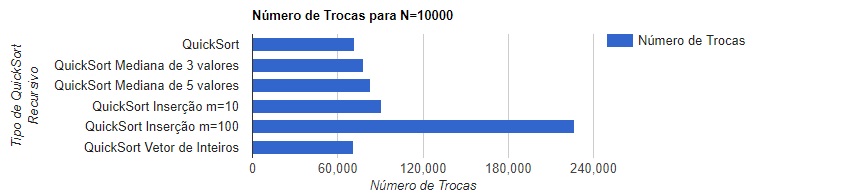




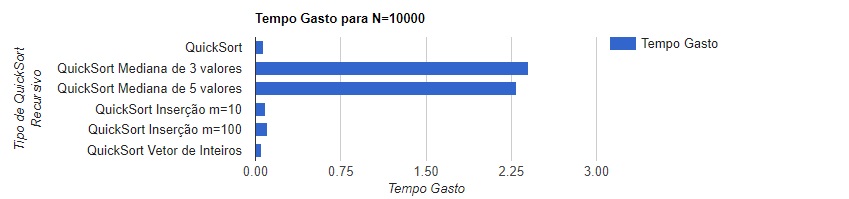




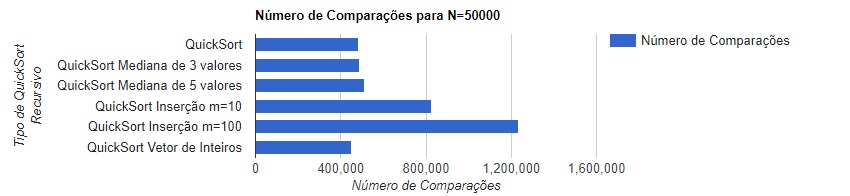




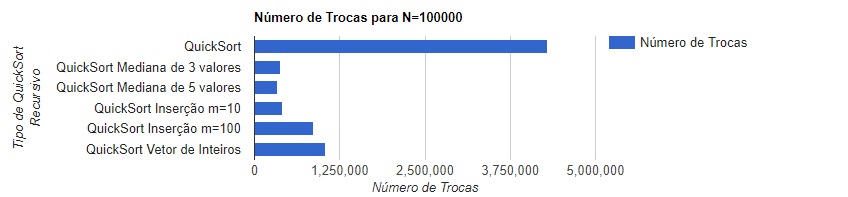


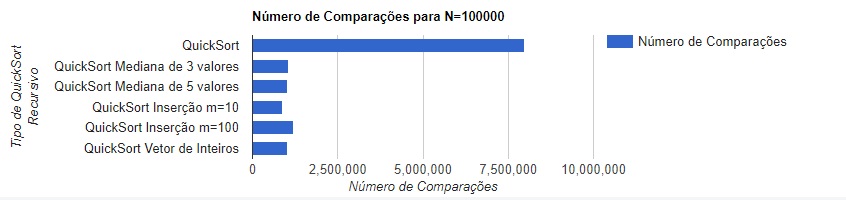






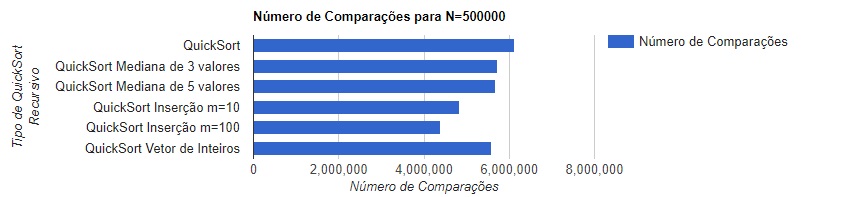


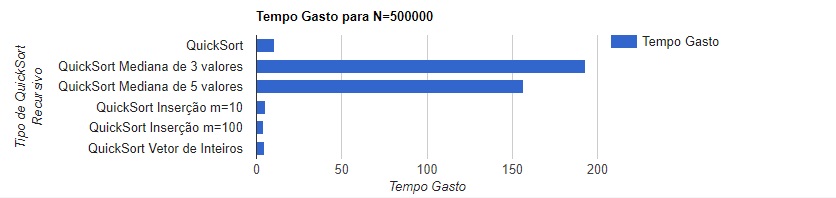


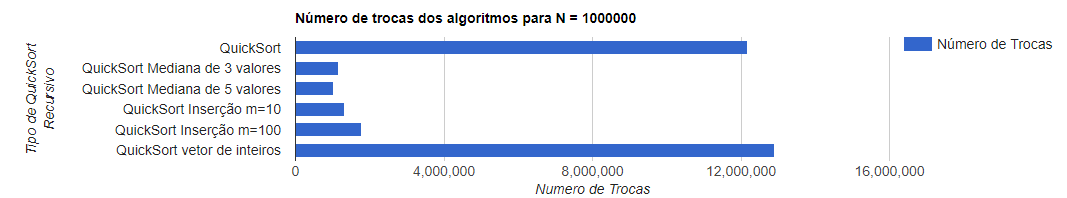


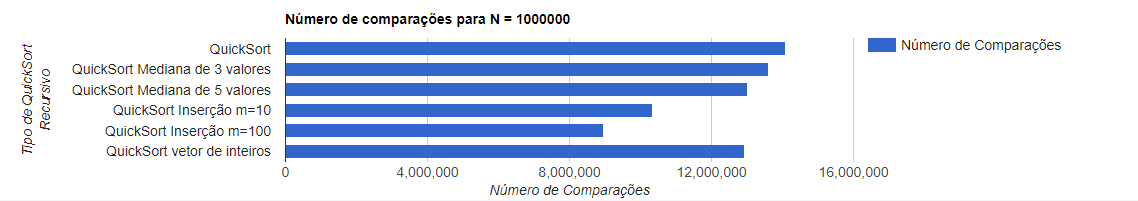














-Análise dos Algoritmos

-Tabelas

As tabelas a seguir foram criadas usando os dados retornados por cada algoritmo de ordenação com N variando entre 1000, 5000, 10000, 50000, 100000, 500000 e 1000000.

De acordo com o desempenho o QuickSort escolhido para comparação foi o Algoritmo QuickSort Recursivo com Insercao com m=10.

Os algoritmos InsertionSort e BubbleSort apresentam alguns dados iguais a 0 a partir de N=100000, isso ocorreu devido ao tamanho do inteiro retornado durante os testes, eles apresentavam valores maiores do que o aceitável pelo tipo de variável int.



















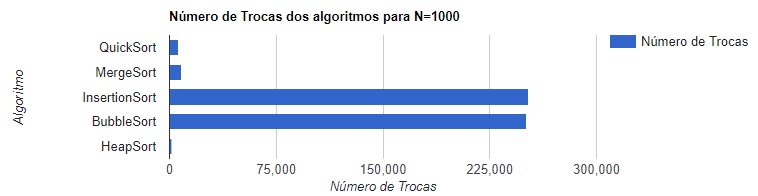


-Gráficos

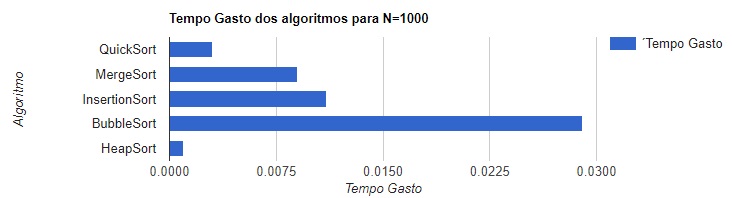
Os gráficos a seguir foram criados usando os dados retornados por cada algoritmo de ordenação com N variando entre 1000, 5000, 10000, 50000, 100000, 500000 e 1000000.

De acordo com o desempenho o QuickSort escolhido para comparação foi o Algoritmo QuickSort Recursivo com Insercao com m=10.

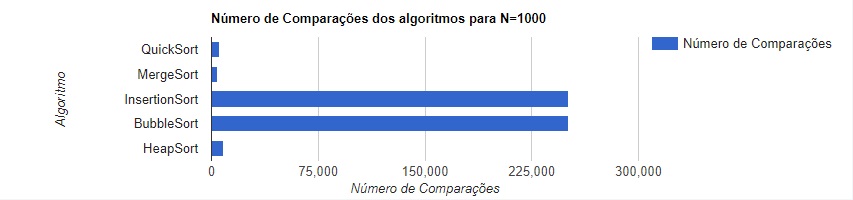
Os algoritmos InsertionSort e BubbleSort apresentam alguns dados iguais a 0 a partir de N=100000, isso ocorreu devido ao tamanho do inteiro retornado durante os testes, eles apresentavam valores maiores do que o aceitável pelo tipo de variável int.

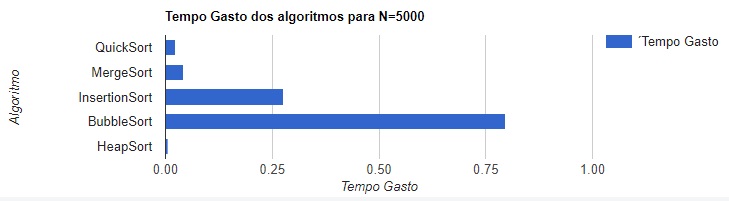


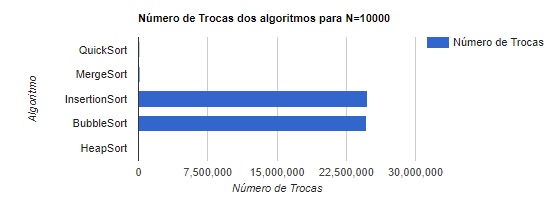


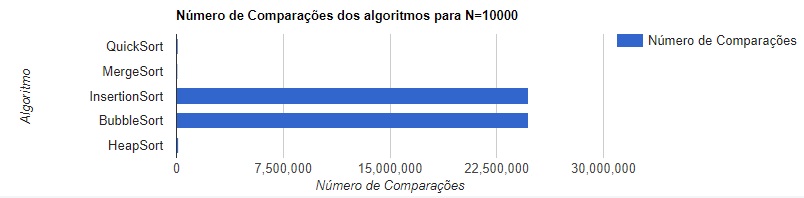


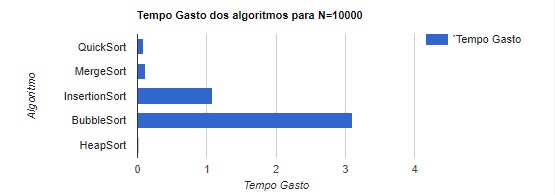
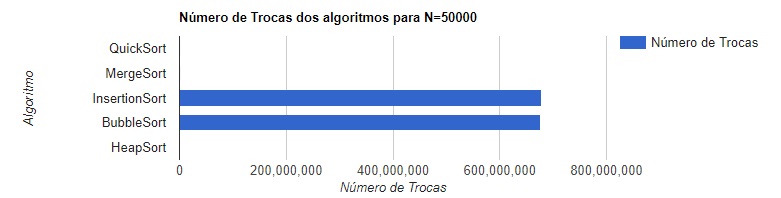


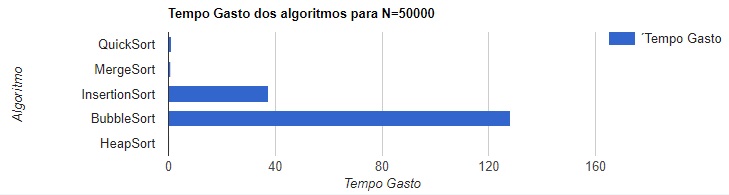


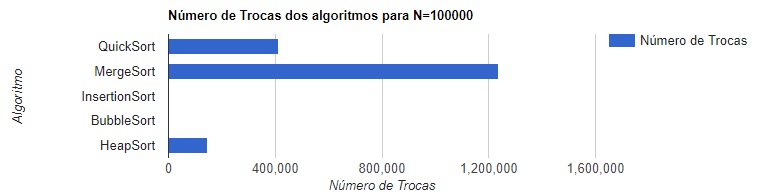


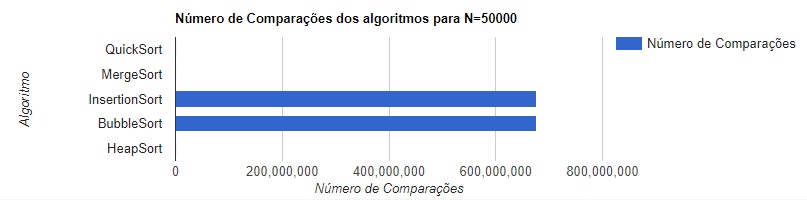


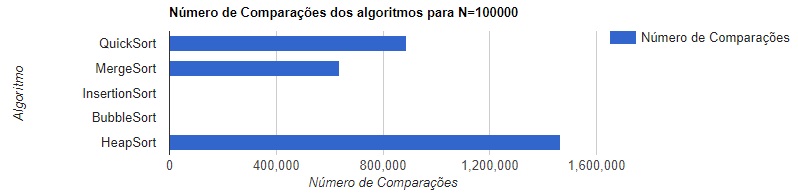


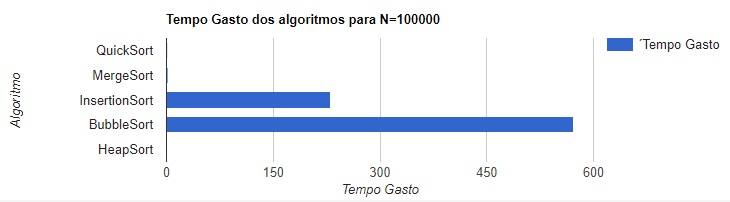




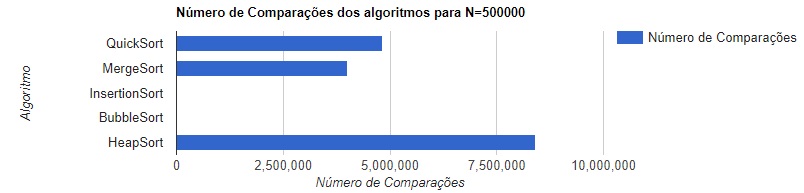
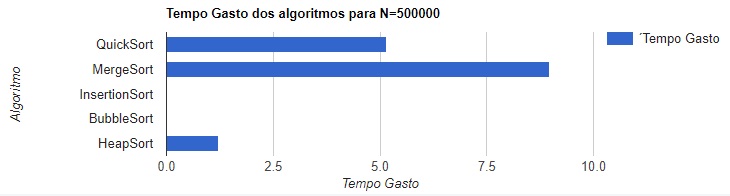






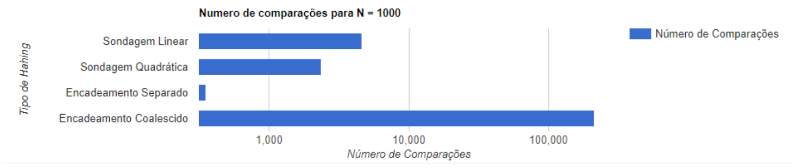


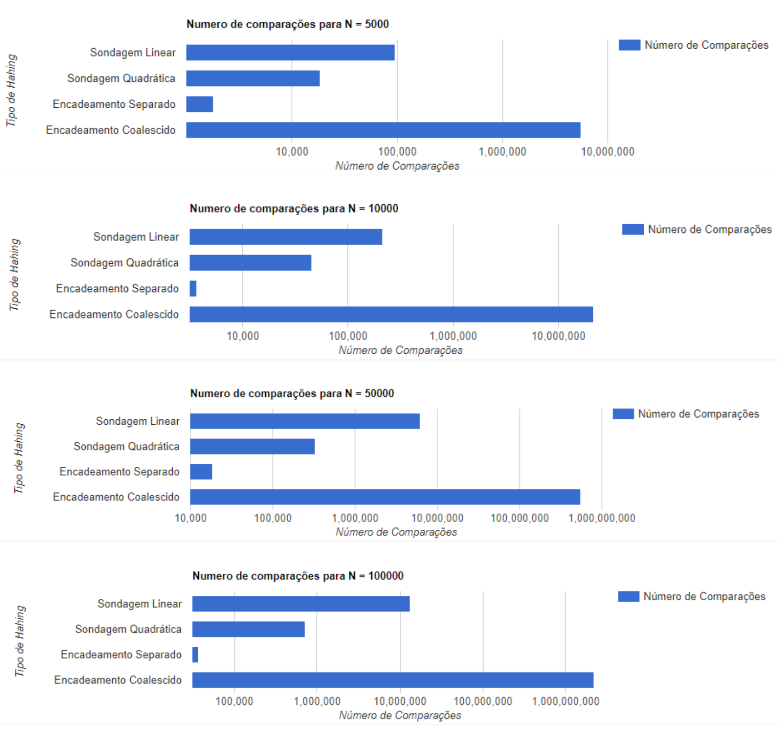


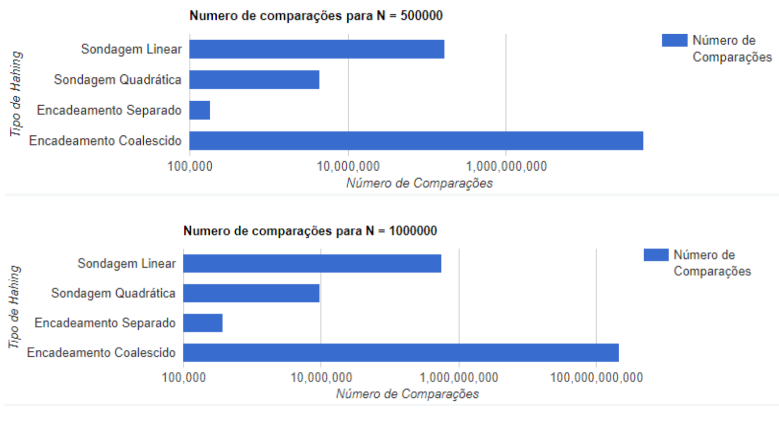


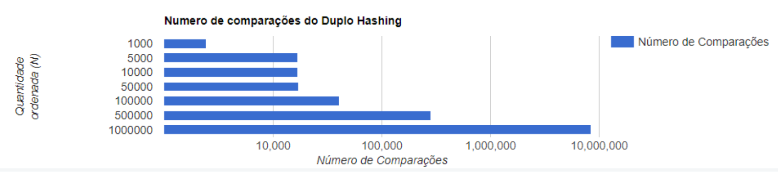












**4 – Análises dos Resultados**

**4.1. Cenário I**

Os tipos de dados comparados foram o Vetor de Objetos do tipo *Tweet* e o vetor de Inteiros. Em termos de trocas e números de comparações, não houve diferença significativa de resultados. Porém, em termos de tempo de execução, a ordenação do Vetor de Inteiros se mostrou mais eficiente, visto que para uma amostra de 1.000.000 entradas o algoritmo gastou 11.039 segundos para finalizar sua execução, tempo significativamente melhor que os 30.161 segundos gastos pelo QuickSort do vetor de Objetos do tipo *Tweet*.

**4.2. Cenário II**

Dentre os diferentes Tipos de QuickSort, o melhor resultado geral obtido foi pelo Algoritmo de QuickSort de Inserção. Para Amostras menores (1.000 e 5.000) não houve diferença significativa entre os resultados, mas à medida em o tamanho da amostra aumentou, o Algoritmo de Inserção com sub-vetores de tamanho menor ou igual a 10 se mostrou mais eficiente. Já com as maiores Amostras (500.000 e 1.000.000), o algoritmo de QuickSort de Inserção sub-vetores de tamanho menor ou igual a 100 se mostrou como o mais eficiente.

O impacto dos diferentes valores de *k* no método de QuickSort com mediana foi principalmente o tempo de execução e gasto em memória, visto que a cada escolha de pivô para cada sub-partição o calculo da mediana era refeito, e o algoritmo aloca memória auxiliar para os k=3 e k=5 valores aleatórios de cada sub-partição.

Já nos algoritmo de QuickSort com Inserção, o impacto entre os diferentes valores de *m,* o algoritmo que utilizava m=100 obteve um pior resultado em número de trocas e comparações, porém se mostrou mais eficiente em tempo de execução.

**4.3. Cenário III**

Considerando os algoritmos de ordenação InsertionSort, MergeSort, HeapSort, QuickSort e BubbleSort, o que obteve melhor resultado foi o algoritmo HeapSort, visto que na maioria das métricas ele obteve o menor número de comparações e trocas, bem como menor tempo de execução. Em alguns casos, onde as amostras eram maiores (500.000 e 1.000.000) algoritmos como InsertionSort e BubbleSort se mostravam inviáveis, visto que o tempo de execução era extremamente alto. Para tais amostras, o algoritmo de QuickSort obteve alto uso de memória, principalmente com a variação utilizando Mediana.

**4.4. Cenário IV**

Os algoritmos de hashing obtiveram resultados significativamente diferentes, e apresentando como melhor desempenho o algoritmo pelo método de Encadeamento Separado, na totalidade dentre as métricas utilizadas. Comparado aos outros algoritmos, o Encadeamento Separado obteve o menor número de comparações, que por vezes, eram inviávelmente maiores (como o Encadeamento Coalescido).

**5 – Dados Sobre o Desenvolvimento**

**5.1 – Hardware e Software Utilizado**

O projeto foi executado em um computador com Core i7 de 3.5GHz, com 16gb de memória RAM, em um sistema operacional Windows 10 Pro de 64bit`s utilizando

a IDE Visual Basic (com linguagem C++). Para a gestão e controle do projeto entre o grupo, a plataforma GitHub foi utilizada.

**5.2 – Divisão de tarefas entre o Grupo**

Bruno Carvalho: Cenario II, Auxílio no Cenario III, Algoritmo de QuickSort por Mediana e Inserção, InsertionSort, funções de escrita em arquivo txt, função de randomização, Menu exibido no console e função de escolha de opções do menu, dentre outras funções do *Main,* execução de bateria de testes em batch dos algoritmos de Ordenação, bem como auxílio na produção do relatório.

Diogo Destefano: Cenario III, Bubble, HeapSort, tabelas e gráficos de resultados de execução, auxílio na produção do relatório.

Rafael Terra: Cenario IV, Funções de Hashing, execução de bateria de testes em batch dos algoritmos de Hash funções de saída em arquivo txt, auxílio na função de carga de Tweets via arquivo txt, função de Hashing da Parte II (Palavras do Momento).

Pedro Bellotti: Cenario I, Auxílio no Cenario III, MergeSort, QuickSort Recursivo, função de carga de Tweets via arquivo txt, função de carga de numero de amostras via arquivo entrada.txt, criação de classes da Parte II (Palavras do Momento), funções de preparação da Parte II, como remoção de espaços extras, remoção de caracteres especiais e separação dos textos dos Tweets em Palavras.