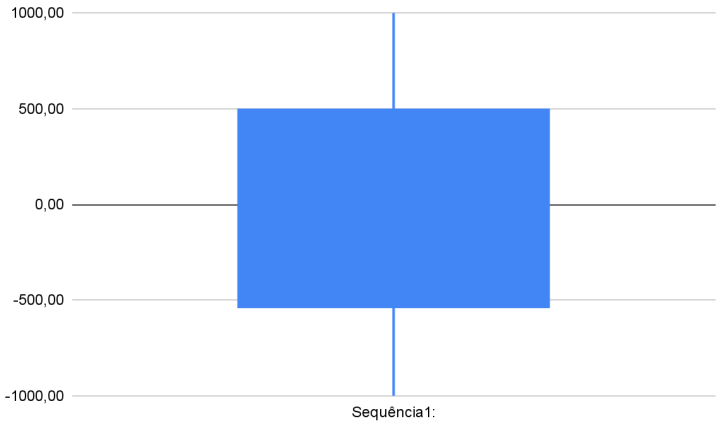
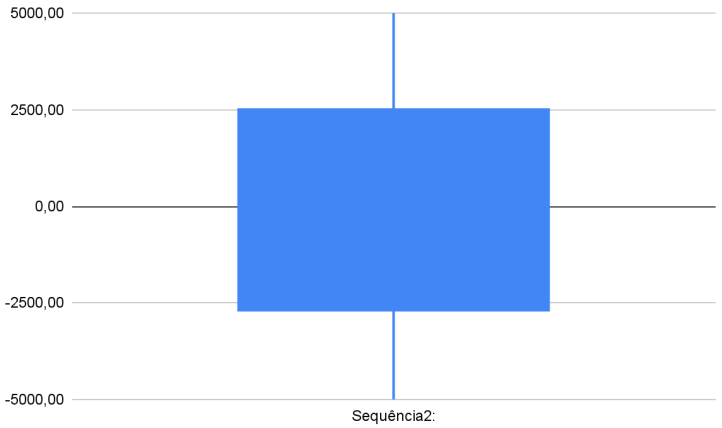
Relatório Atividade em Grupo 1
Algoritmo e Estrutura de Dados 2
Nome: Guilherme Silva do Nascimento
RA:156554

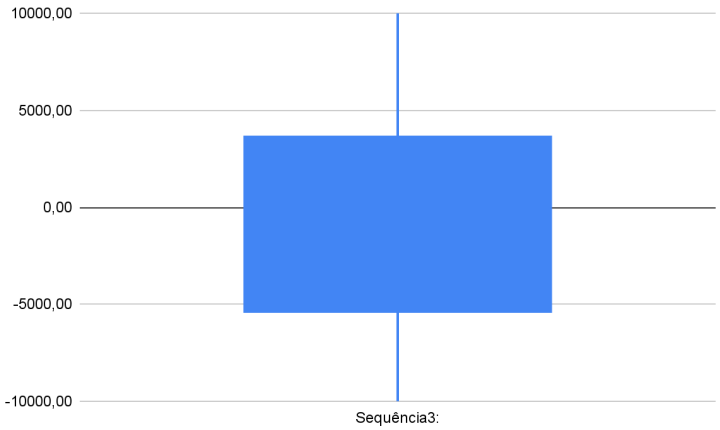
Neste trabalho foram implementados 3 diferentes algoritmos de ordenação(SelectionSort, QuickSort, BucketSort), com uma ressalva a ser discutida posteriormente, na linguagem C++20 pelo editor Visual Studio Code.

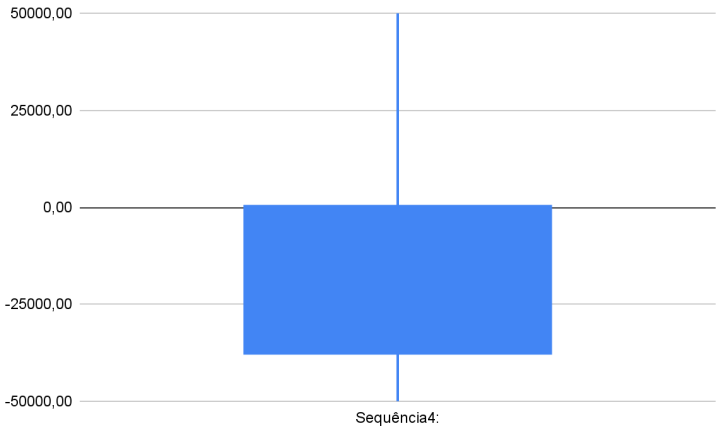
Cada algoritmo foi repetido três vezes para cada sequência de números aleatórios de tamanhos 1000, 5000, 10000, 50000 e 100000 geradas de forma aleatória e uniforme, onde cada elemento varia dentro do intervalo [-tamanho, +tamanho] com números inteiros e reais(código disponível no arquivo GeraEntradas.cpp), após a criação da sequência, a foi embaralhada utilizando uma variação do algoritmo de Fisher-Yates desenvolvida por Richard Durstenfeld. Então, cada sequência é escrita em um arquivo .csv(disponíveis na pasta Entradas), contudo, por motivos práticos de implementação e para facilitar a visualização, todos os números foram convertidos para o tipo “float” e, são escritos com apenas 3 casas decimais, com isso, é perdida precisão nos números ao ser feita a leitura dos arquivos. Porém, para o intuito deste experimento, isso não será relevante, visto que, não se espera que valores específicos influenciem na execução dos algoritmos.

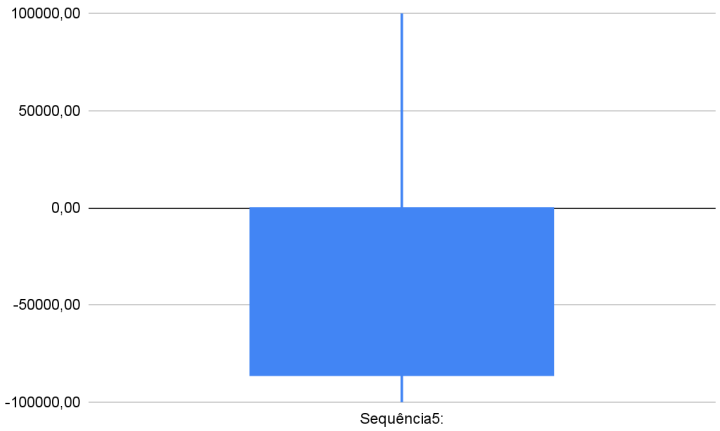
Segue BoxPlots das Sequências obtidas:











É notória a tendência aos números negativos, nas sequências 4 e 5, porém ela é dada pela natureza aleatória da geração de número. Além disso, como dito anteriormente, os valores específicos de cada elemento não são relevantes para este experimento.

A estrutura dos códigos para a execução de todos os algoritmos consiste em:

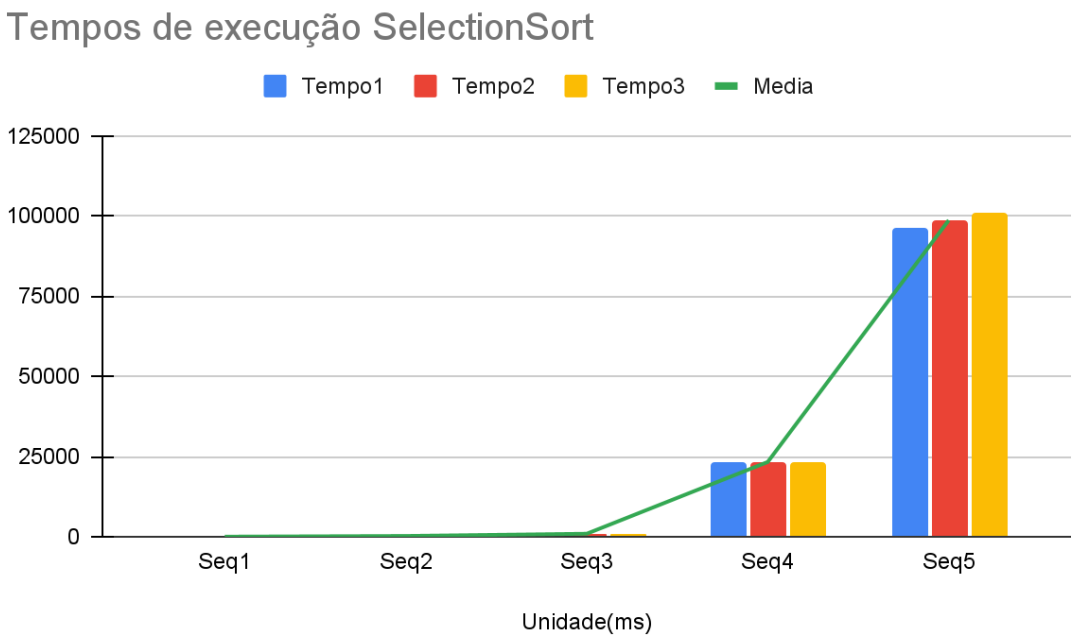
1) Definir o caminho pelo qual os arquivos de entrada serão lidos para onde os de saída serão escritos
2) Realizar a leitura do arquivo de entrada

3) Executar algoritmo de ordenação e fazer a leitura de seu tempo de execução
4) Escrever em um arquivo .csv o tempo computado
5) Escrever a sequência ordenada em um arquivo de saída .csv em seu respectivo diretório
6) Repetir a partir de 2) 3 vezes
7) Escrever a média dos tempos no mesmo arquivo em que os tempos de execução foram escritos
8) Repetir a partir de 2) para cada arquivo de entrada

1. SelectionSort

Tanto o conceito quanto a implementação do algoritmo de SelectionSort são simples e intuitivos, apenas percorremos todo o vetor em busca do, neste caso, o menor elemento e então o colocamos na primeira posição, avançamos uma posição e repetimos o processo.

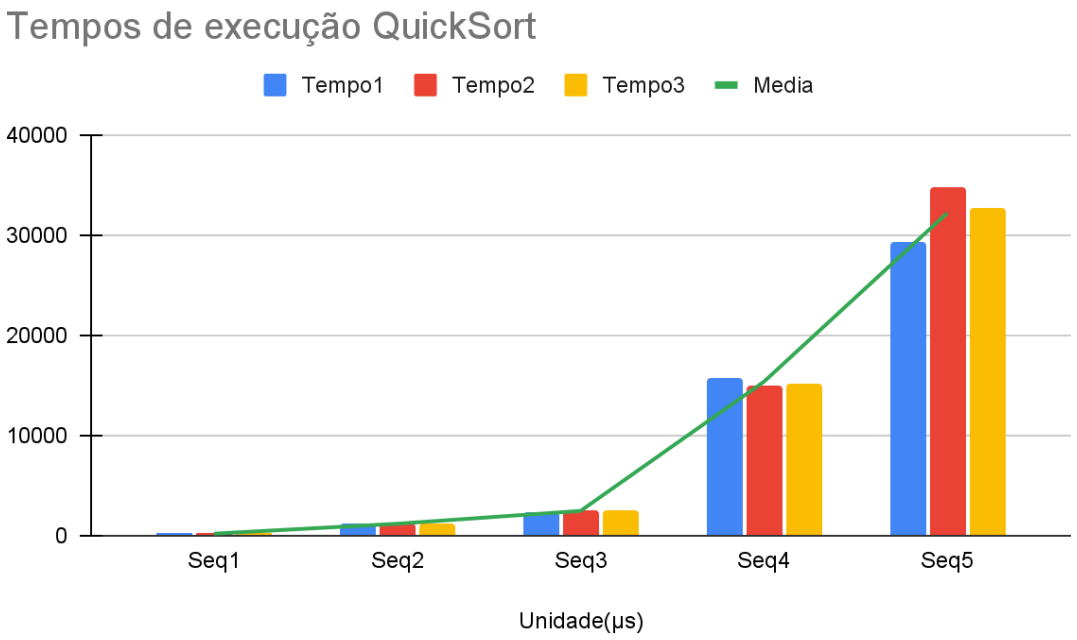
Este método acaba por ser bastante custoso em tempo de execução, pois é necessário fazer todo o percorrimento do vetor para cada posição que se deseja ordenar, em outras palavras possui um tempo de execução $O(n^2)$, o que o torna um algoritmo não muito eficiente ao custo de sua simples implementação.



2. QuickSort

Apesar de conceitualmente o algoritmo de QuickSort ser um pouco mais complexo, sua implementação continua sendo relativamente simples(sendo definida por recursividade). Fundamentalmente o QuickSort revolve em torno de um pivô(definido neste projeto como o último elemento do vetor) que é colocado em um ponto do vetor no qual em índices anteriores só há valores menores que ele e em índices posteriores, valores maiores. Com sua implementação recursiva é notável que o algoritmo faz uso da ideia de “dividir e conquistar” separando, através do pivô, os vetores em dois subvetores nos quais é realizado o mesmo processo.

Seguindo com a ideia de um algoritmo de fácil visualização e compreensão o QuickSort acaba por ser consideravelmente mais eficiente que o SelectionSort, visto que ao quebrar o vetor em partes menores a cada iteração reduz-se significativamente a quantidade de comparações necessária ao fim do algoritmo com um tempo de execução médio de O(nlog(n)). Contudo a eficiência do QuickSort é bastante dependente do método pelo qual o pivô é definido, sendo no seu pior caso $O(n^2)$, para este experimento o método escolhido foi um dos mais simples e não o que seria considerado como eficiente, porém para meios de análise do algoritmo sem adicionar comparações para a escolha de pivô, se faz suficiente.



3. BucketSort

Em conceito o algoritmo de BucketSort já exige um nível de interpretação e visualização um pouco maior do que os algoritmos anteriores, Sua implementação, em certos casos, também acaba por ser mais complexa. Seu funcionamento se baseia em adicionar os valores em algo que seria equivalente a uma hashtable onde a hash key é definida através do valor de suas casas decimais e o tratamento de colisões mantém a mesma hashkey para o elemento, neste trabalho todos os valores foram convertidos para o intervalo [0,1], e então, é inserido em seu respectivo “balde”(a quantidade de baldes é dependente da implementação, neste experimento foi definido como 10). Após toda a inserção nos baldes, eles são individualmente ordenados por outro algoritmo qualquer(normalmente utiliza-se InsertionSort, como neste trabalho) e então, balde a balde, são concatenados os valores ordenados em um único vetor.

Devido a, muito possivelmente, inexperiência do autor tanto com a linguagem de programação utilizada quanto com o manuseio de arquivos, houve dificuldade para a implementação do BucketSort, de modo que não foi possível testar de forma empírica a eficiência do algoritmo que varia diretamente com o tamanho de cada balde, podendo ir de O(n) até $O(n^2)$(para este projeto era esperado O(n)).