

**TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS**

**ISABELLA CAMILLI MELETANI**

**ÍCARO GABRIEL PEREIRA CARVALHO**

**RELATÓRIO PARA A DISCIPLINA DE ALGORITMOS**

**SOBRE ORDENAÇÃO EM C E JAVA**

Natal – RN

2025

**RESUMO**

Reumo remuremungid

**SUMÁRIO**

**1. INTRODUÇÃO AOS ALGORITMOS DE ORDENAÇÃO**

**2. DESCRIÇÃO DA IMPLEMENTAÇÃO DE CADA ALGORITMO**

**3. METODOLOGIA UTILIZADA PARA OS EXPERIMENTOS**

**4. APRESENTAÇÃO DOS RESULTADOS**

**4.1. BUBBLE SORT**

**4.1.1. C**

**4.1.1.1. Implementação**

**4.1.1.2. Gráfico**

**4.1.1.3 Análise**

**4.1.2. Java**

**4.1.2.1. Implementação**

**4.1.2.2 Gráfico**

**4.1.2.3 Análise**

**4.1.3. Comparação**

**1. INTRODUÇÃO AOS ALGORITMOS DE ORDENAÇÃO**

O problema de ordenar dados de maneira eficiente é uma questão fundamental que surgiu cedo na história da computação, à medida que os primeiros computadores começaram a ser utilizados para processar informações em grande escala. Durante as primeiras décadas da computação, a necessidade de ordenar dados de forma rápida e eficaz se tornou cada vez mais evidente, especialmente à medida que as empresas e organizações começaram a lidar com volumes crescentes de dados. No início, os computadores eram usados principalmente para realizar cálculos matemáticos e resolver problemas científicos, mas logo se percebeu que muitos processos requeriam a organização de grandes quantidades de dados, seja para análise, armazenamento ou busca. A ordenação eficiente de dados passou, então, a ser uma tarefa central em diversas áreas, como bancos de dados, sistemas de arquivos e pesquisas em grandes volumes de dados.

Com o aumento da capacidade de armazenamento e o surgimento de sistemas mais complexos de informação, a computação começou a enfrentar o desafio de lidar com dados em grande escala. Em um cenário com milhões ou bilhões de dados a serem organizados, a simples comparação de elementos e a troca entre eles de maneira ineficiente não eram mais viáveis. Dessa forma, surgiu a necessidade de criar algoritmos que pudessem ordenar grandes volumes de dados de forma mais rápida e com menos consumo de recursos computacionais. Esse desafio levou ao desenvolvimento de algoritmos de ordenação, que têm como objetivo reduzir o tempo necessário para ordenar grandes quantidades de dados, utilizando diferentes técnicas como divisão e conquista, ordenação por troca, entre outras.

Nos anos de 1940 e 1950, à medida que os primeiros computadores eletrônicos começaram a ser utilizados, surgiu a necessidade urgente de organizar dados de forma mais eficiente. Os computadores daquela época eram limitados em termos de capacidade de processamento e armazenamento, mas mesmo assim, as aplicações começaram a demandar uma maneira de organizar, classificar e buscar informações de maneira mais eficaz. Esse cenário gerou o desenvolvimento de algoritmos simples para resolver o problema de ordenação, sendo o Bubble Sort e o Selection Sort dois dos primeiros a serem propostos.

O Bubble Sort e o Selection Sort foram entre os primeiros algoritmos a serem criados para ordenar listas de dados, mas, apesar de sua simplicidade e facilidade de implementação, eles não são eficientes para grandes volumes de dados. O Bubble Sort, por exemplo, funciona repetidamente trocando elementos adjacentes que estão fora de ordem, o que resulta em uma complexidade de tempo O(n²) no pior caso, tornando-o lento à medida que o número de elementos a ser ordenado cresce. Da mesma forma, o Selection Sort, que seleciona repetidamente o menor (ou maior) elemento da lista e o coloca na posição correta, também apresenta a mesma complexidade O(n²) no pior caso, o que o torna ineficiente quando comparado a métodos mais avançados de ordenação.

Esses algoritmos, embora simples e intuitivos, foram marcos importantes na evolução das

soluções para o problema de ordenação. Eles representaram os primeiros passos na criação de soluções computacionais, mas também evidenciaram as limitações de abordagens baseadas apenas em trocas sequenciais de dados. O desenvolvimento desses algoritmos ajudou a perceber a necessidade de métodos mais sofisticados e eficientes para lidar com volumes cada vez maiores de dados, impulsionando o avanço da pesquisa e desenvolvimento de algoritmos mais rápidos e com complexidade de tempo mais favorável.

Na década de 1960 e 1970, o desenvolvimento de algoritmos de ordenação passou por uma revolução com a introdução de métodos mais eficientes, como o Merge Sort e o Quick Sort. Esses novos algoritmos basearam-se em técnicas avançadas, como divisão e conquista, o que lhes permitiu superar os algoritmos anteriores em termos de velocidade e eficiência, especialmente ao lidar com grandes volumes de dados.

O Merge Sort, proposto por John von Neumann em 1945, ganhou popularidade na década de 1960 devido à sua capacidade de ordenar grandes quantidades de dados de maneira eficiente. O Merge Sort possui uma complexidade de tempo de O(n log n), o que representa uma grande melhoria em relação aos algoritmos anteriores como o Bubble Sort. O seu desempenho é garantido mesmo no pior caso, o que o torna particularmente útil para grandes conjuntos de dados e para situações em que a estabilidade da ordenação é importante.

O Quick Sort, embora tenha uma complexidade média de O(n log n), é conhecido por sua excelente performance em muitos casos práticos, tornando-se um dos algoritmos de ordenação mais rápidos e amplamente utilizados, especialmente quando se trata de dados não totalmente ordenados. Seu desempenho no pior caso é O(n²), mas, com o uso de técnicas como escolha aleatória de pivôs ou particionamento balanceado, seu comportamento tende a ser muito bom na maioria das situações.

Ambos os algoritmos representaram uma verdadeira revolução no campo da ordenação de dados. Ao aplicar a técnica de divisão e conquista, eles conseguiram reduzir significativamente o tempo de execução, tornando-se soluções muito mais eficientes para problemas de ordenação de grandes volumes de dados. A introdução desses algoritmos marcou um ponto de virada na pesquisa sobre eficiência algorítmica, incentivando o desenvolvimento de novas abordagens e melhorias na teoria dos algoritmos de ordenação.

Nas décadas de 1980 e 1990, com a popularização dos computadores pessoais e o crescimento exponencial do volume de dados gerados, a ordenação de informações tornou-se uma tarefa essencial em muitos sistemas computacionais. À medida que as tecnologias avançavam e os computadores se tornavam mais acessíveis, o armazenamento e a manipulação de grandes quantidades de dados passaram a ser uma necessidade comum em diversos setores, como bancos de dados, sistemas de arquivos, redes e aplicativos empresariais. A ordenação eficiente desses dados se tornou crucial não apenas para melhorar o desempenho de sistemas, mas também para permitir a realização de tarefas complexas de busca e recuperação de informações, além de otimizar processos de análise de dados.

A maior compreensão da complexidade computacional possibilitou o desenvolvimento de

novas abordagens para otimizar a ordenação de dados. Algoritmos como Quick Sort, Merge Sort e Heap Sort tornaram-se ainda mais populares e amplamente utilizados, pois eram mais rápidos e eficientes do que os métodos tradicionais. Além disso, nesse período, houve a adoção crescente de técnicas de ordenação paralela e distribuída, que aproveitavam a multiplicação de núcleos de processamento e sistemas de computação em rede para acelerar o processo de ordenação em grandes volumes de dados. A crescente preocupação com a escabilidade dos algoritmos de ordenação também refletiu a necessidade de soluções adequadas para o processamento de dados em um mundo digital cada vez mais interconectado e dependente de informações.

A partir da década de 1990 e ao longo do período presente, os algoritmos de ordenação passaram a ganhar uma importância ainda mais crítica, especialmente com o aumento da ascensão da internet e o crescimento exponencial da quantidade de dados gerados em nível global. Com o advento de grandes volumes de informações em diversos setores, como redes sociais, comércio eletrônico e pesquisas científicas, a ordenação de dados tornou-se uma tarefa essencial para garantir a eficiência de sistemas que dependem de dados ordenados para pesquisa, análise e apresentação de informações. O aumento da quantidade e complexidade dos dados impulsionou a criação de soluções cada vez mais sofisticadas e escaláveis.

Uma das inovações mais notáveis nesse período foi a criação do Tim Sort, um algoritmo de ordenação híbrido desenvolvido por Tim Peters em 2002 para a linguagem de programação Python. O Tim Sort é uma combinação do Merge Sort e do Insertion Sort, aproveitando as vantagens de ambos. Ele foi projetado para aproveitar a estrutura de dados parcialmente ordenada, que é comum em dados reais, o que o torna especialmente eficiente em muitos casos práticos.

Essas inovações refletem a adaptação dos algoritmos de ordenação às necessidades de processamento em larga escala, com foco na eficiência e na capacidade de lidar com volumes massivos de dados. À medida que a tecnologia continua a evoluir, a necessidade de soluções de ordenação rápidas e escaláveis só tende a crescer, com o surgimento de novas abordagens e a evolução das técnicas de ordenação paralela e distribuída, adaptando-se ao novo cenário de dados massivos e processamento em tempo real.

**2. DESCRIÇÃO E IMPLEMENTAÇÃO DE CADA ALGORITMO**

O Bubble Sort é um algoritmo de ordenação simples, mas ineficiente para listas grandes. Ele funciona comparando elementos adjacentes de uma lista e, se os elementos não estão na ordem correta, troca-os de lugar. Esse processo de comparação e troca é repetido até que a lista esteja completamente ordenada. Em outras palavras, o algoritmo percorre a lista várias vezes, comparando e trocando os elementos até que não sejam mais necessárias trocas. A cada passagem, o maior elemento "borbulha" para o final da lista, o que dá origem ao nome do algoritmo.

Por exemplo, dada a lista [5, 2, 9, 1, 5, 6], o Bubble Sort começa comparando o primeiro elemento (5) com o segundo (2). Como 5 é maior que 2, os dois elementos são trocados, resultando na lista [2, 5, 9, 1, 5, 6]. O processo de comparação e troca continua, agora com 5 e 9, mas como 5 é menor que 9, nada acontece. O algoritmo segue esse processo para os elementos seguintes e repete essa sequência de passos até que todos os elementos estejam na ordem crescente desejada.

Em termos de complexidade teórica, o Bubble Sort apresenta uma performance que varia conforme o estado inicial da lista. No melhor caso (quando a lista já está ordenada), o algoritmo realiza apenas uma passagem pela lista, verificando que nenhum elemento precisa ser trocado. Nesse caso, a complexidade é O(n), onde n é o número de elementos da lista. No pior caso, que ocorre quando a lista está em ordem inversa, o algoritmo precisa percorrer toda a lista e realizar trocas em cada comparação, resultando em uma complexidade de O(n²). Esse comportamento é típico do Bubble Sort, e sua ineficiência se torna evidente em listas grandes, o que limita seu uso em situações que exigem desempenho elevado.  
  
O Insertion Sort é um algoritmo de ordenação que funciona de maneira semelhante ao processo de ordenar cartas em mãos. O algoritmo começa com o segundo elemento da lista e o compara com os elementos anteriores. Se o elemento comparado for menor que o anterior, ele é "inserido" na posição correta, movendo os elementos maiores para a direita para abrir espaço. Esse processo é repetido para cada elemento subsequente, até que a lista esteja completamente ordenada. A ideia central é que, a cada iteração, a parte da lista que já foi analisada estará sempre ordenada.

Por exemplo, dada a lista [5, 2, 9, 1, 5, 6], o algoritmo começa com o segundo elemento, 2, e o compara com o primeiro, 5. Como 2 é menor que 5, o algoritmo move o 5 para a direita e insere o 2 na posição correta, resultando na lista [2, 5, 9, 1, 5, 6]. Agora, o algoritmo avança para o próximo elemento, que é 9. Ele compara 9 com 5 e, como 9 é maior, não há troca e o 9 permanece na mesma posição. O processo segue para o 1, que é comparado com os elementos anteriores. O 1 é menor que 5 e 2, então o algoritmo move esses elementos para a direita, inserindo o 1 na posição inicial da lista. O resultado intermediário é [1, 2, 5, 9, 5, 6]. O algoritmo continua com o próximo 5, que é comparado com 9. Como o 5 é menor, o 9 é movido para a direita, e o 5 é inserido antes do 9. A lista se torna [1, 2, 5, 5, 9, 6]. Por fim, o

algoritmo compara o 6 com 9, trocando-os para inserir o 6 na posição correta, resultando na lista final ordenada: [1, 2, 5, 5, 6, 9]. Esse processo continua repetidamente até que todos os elementos estejam na posição correta.

Em termos de complexidade teórica, o Insertion Sort tem desempenho variável dependendo da ordenação inicial dos dados. No melhor caso, quando a lista já está ordenada, o algoritmo apenas verifica que não há necessidade de mover nenhum elemento, resultando em uma complexidade de O(n), onde n é o número de elementos da lista. No pior caso, que ocorre quando a lista está em ordem inversa, o algoritmo precisa mover cada elemento para a posição correta, realizando uma série de trocas e comparações, o que resulta em uma complexidade de O(n²). Portanto, embora o Insertion Sort seja eficiente para listas pequenas ou parcialmente ordenadas, sua complexidade quadrática limita seu uso em listas maiores.

O Selection Sort é um algoritmo de ordenação que funciona identificando o menor (ou maior, dependendo da implementação) elemento de uma lista e trocando-o com o primeiro elemento não ordenado. O processo é repetido para os próximos elementos, até que toda a lista esteja ordenada. A principal característica desse algoritmo é que, a cada iteração, ele seleciona o valor mínimo e coloca-o na posição correta, movendo gradualmente os menores elementos para o início da lista.

Por exemplo, dada a lista [5, 2, 9, 1, 5, 6], o algoritmo começa comparando todos os elementos para encontrar o menor valor. O menor número é 1, então o algoritmo troca 1 com o primeiro elemento, 5, resultando na lista [1, 2, 9, 5, 5, 6]. Em seguida, o algoritmo encontra o menor valor na parte restante da lista, que é 2, e, como ele já está na posição correta, não há troca. O algoritmo continua procurando o menor número na sublista [9, 5, 5, 6], encontra o 5 e troca-o com o 9. O processo continua até que todos os elementos sejam colocados em suas posições corretas, resultando na lista final ordenada [1, 2, 5, 5, 6, 9].

Em termos de complexidade teórica, o Selection Sort tem desempenho O(n²) tanto no melhor quanto no pior caso. No melhor caso, mesmo que a lista já esteja ordenada, o algoritmo ainda percorre toda a lista em busca do menor elemento, realizando comparações desnecessárias. No pior caso, que ocorre quando a lista está em ordem inversa, o algoritmo também percorre toda a lista a cada iteração, trocando elementos para colocar o menor valor na posição correta. Por isso, o Selection Sort não é muito eficiente para listas grandes, embora tenha a vantagem de usar uma quantidade mínima de memória extra, já que as trocas são feitas no próprio local dos elementos.

O Merge Sort é um algoritmo de ordenação que se baseia na técnica de divisão e conquista. A técnica de divisão e conquista consiste em dividir um problema grande em subproblemas menores e mais fáceis de resolver, resolver esses subproblemas e, finalmente, combinar as soluções de maneira eficiente. No caso do Merge Sort, o algoritmo divide recursivamente a lista em duas metades, até que cada sublista contenha apenas um elemento, e então começa o processo de combinação dessas sublistas de maneira ordenada.

Por exemplo, dada a lista [5, 2, 9, 1, 5, 6], o Merge Sort começa dividindo a lista ao meio,

criando duas sublistas: [5, 2, 9] e [1, 5, 6]. Essas sublistas são novamente divididas até que cada uma contenha apenas um único elemento: [5], [2], [9] e [1], [5], [6]. A seguir, as sublistas são mescladas. Primeiro, as sublistas [5] e [2] são mescladas para formar [2, 5], e as sublistas [9] e [5] são mescladas para formar [5, 9]. Em seguida, as duas sublistas resultantes, [2, 5] e [5, 9], são mescladas para formar [2, 5, 5, 9]. Da mesma forma, as sublistas [1] e [5] são mescladas para formar [1, 5], e as sublistas [6] e [1, 5] são mescladas para formar [1, 5, 6]. Finalmente, as duas sublistas ordenadas [2, 5, 5, 9] e [1, 5, 6] são mescladas para formar a lista final ordenada [1, 2, 5, 5, 6, 9].

Em termos de complexidade teórica, o Merge Sort tem uma performance de O(n log n) tanto no melhor quanto no pior caso. No melhor caso, quando os dados já estão quase ordenados, o algoritmo ainda divide a lista em partes e mescla as sublistas de maneira eficiente. No pior caso, mesmo quando a lista está em ordem inversa, o algoritmo ainda mantém essa eficiência, dividindo as listas de forma balanceada e mesclando-as de maneira ordenada. O desempenho consistente do Merge Sort o torna muito eficiente para grandes volumes de dados, especialmente em cenários onde a estabilidade (manter a ordem relativa dos elementos iguais) é necessária. No entanto, uma desvantagem do Merge Sort é que ele exige espaço adicional para armazenar as sublistas temporárias durante o processo de mesclagem.

O Quick Sort é um algoritmo de ordenação eficiente baseado na técnica de divisão e conquista. O funcionamento do Quick Sort envolve a escolha de um elemento chamado pivô e a partição da lista em duas sublistas: uma contendo elementos menores que o pivô e outra com elementos maiores. Em seguida, o processo é recursivamente repetido para essas sublistas, até que toda a lista esteja ordenada. O Quick Sort é altamente eficiente porque, ao dividir a lista em sublistas menores, ele reduz o problema original a partes menores e mais fáceis de resolver.

Por exemplo, dada a lista [5, 2, 9, 1, 5, 6], o algoritmo começa escolhendo um pivô, como o número 5. A lista é então reorganizada de forma que todos os elementos menores que 5 fiquem à esquerda do pivô e todos os elementos maiores que 5 fiquem à direita. Após a partição, a lista fica assim: [2, 1, 5, 5, 9, 6]. Agora, o algoritmo aplica a mesma técnica recursivamente às duas sublistas [2, 1] e [5, 9, 6]. Para a sublista [2, 1], o número 1 é escolhido como pivô e a lista é reorganizada para [1, 2]. Para a sublista [5, 9, 6], o número 5 é escolhido como pivô, e a lista é reorganizada para [5, 6, 9]. Finalmente, o algoritmo retorna a lista ordenada [1, 2, 5, 5, 6, 9].

Em termos de complexidade teórica, o Quick Sort tem desempenho de O(n log n) no melhor caso, que ocorre quando a escolha do pivô divide a lista de maneira balanceada, ou seja, cada divisão resulta em sublistas de tamanho aproximadamente igual. Esse caso ocorre quando o pivô está em uma posição central. No pior caso, que acontece quando o pivô escolhido é o menor ou maior elemento da lista, as partições são extremamente desbalanceadas, o que leva o algoritmo a se comportar como uma simples busca sequencial. Nesse caso, a complexidade se torna O(n²), o que torna o algoritmo significativamente mais lento. Para mitigar o pior caso, versões aprimoradas do Quick Sort, como o Quick Sort aleatório, escolhem o pivô de forma

aleatória para aumentar a probabilidade de divisões balanceadas, melhorando a eficiência geral.

**3. METODOLOGIA UTILIZADA PARA OS EXPERIMENTOS**

A metodologia utilizada para os experimentos no trabalho foi projetada com o objetivo de comparar o desempenho dos algoritmos de ordenação implementados nas linguagens C e Java. O foco estava na medição do tempo de execução para diferentes entradas, com o intuito de analisar a eficiência dos algoritmos em diferentes cenários. Para isso, o código foi estruturado de forma que a ordenação fosse realizada dentro de uma função específica. A partir dessa estrutura, a parte principal do código se encarregava de abrir os arquivos de entrada, medir o tempo de execução antes e depois da chamada da função de ordenação, e gerar um arquivo de saída contendo os tempos de execução para cada conjunto de dados.

A geração dos dados de entrada foi feita por meio de um script em Python, que criava arquivos contendo números inteiros aleatórios. Os arquivos foram gerados em três diferentes tipos de ordenação: desordenada, ordenada e inversamente ordenada. Isso permitiu testar o desempenho dos algoritmos em cenários representando o melhor caso (lista já ordenada), o pior caso (lista ordenada de forma inversa) e o caso médio (lista desordenada). Para cada tamanho de entrada (100, 1.000, 10.000, 100.000 e 1.000.000 de números), os arquivos foram gerados e utilizados nos experimentos para testar a eficiência dos algoritmos.

A medição do tempo de execução foi realizada utilizando as bibliotecas de tempo oferecidas pelas linguagens C e Java. A estratégia adotada envolveu gravar em variáveis o tempo antes da execução da função de ordenação e o tempo após sua conclusão. A diferença entre esses valores forneceu o tempo de execução específico para cada chamada do algoritmo de ordenação. Esse processo foi repetido quatro vezes para cada conjunto de dados, e a média dos tempos foi calculada para gerar uma análise mais confiável e precisa dos resultados. Essa abordagem possibilitou a comparação dos algoritmos em diferentes cenários de entradas e também possibilitou o cálculo da média dos tempos de execução.

Para a visualização dos resultados, foram gerados gráficos que representaram o tempo de execução dos algoritmos em função do tamanho da entrada e do tipo de ordenação dos dados. Diversos gráficos foram produzidos para comparar o desempenho em diferentes condições, o que permitiu uma análise visual clara e objetiva das diferenças entre os algoritmos. Esses gráficos foram fundamentais para ilustrar as variações de desempenho em cada cenário e ajudar na interpretação dos dados obtidos.

Por fim, a análise crítica dos resultados obtidos permitiu comparar os algoritmos de ordenação em termos de eficiência, destacando os algoritmos mais rápidos e os mais lentos em diferentes situações. A análise levou em consideração tanto a complexidade teórica dos algoritmos quanto os resultados práticos observados nos experimentos, fornecendo uma visão abrangente sobre a eficácia de cada algoritmo em cenários reais de uso.

**4. APRESENTAÇÃO DOS RESULTADOS**

**4.1. BUBBLE SORT**

**4.1.1. C**

**4.1.1.1. Implementação**

****

Na parte principal do código, na linha 17, se define a variavel entradas[], sendo este definido com 5 valores: 100, 1000, 10000, 100000 e 1000000. Esses valores representam os tamanhos dos arrays que o código vai processar. Para cada valor de entradas[i], será gerado um nome de arquivo específico e lido um conjunto de dados correspondente a esse tamanho de array.

Na linha 19 é iniciado o loop externo, que será executado 4 vezes, no intuito de repetir 4 vezes todos os testes par que assim seja calculada a média. Dentro do loop externo está presente outro loop, este de tamanho 5 para percorrer todo o array entradas.

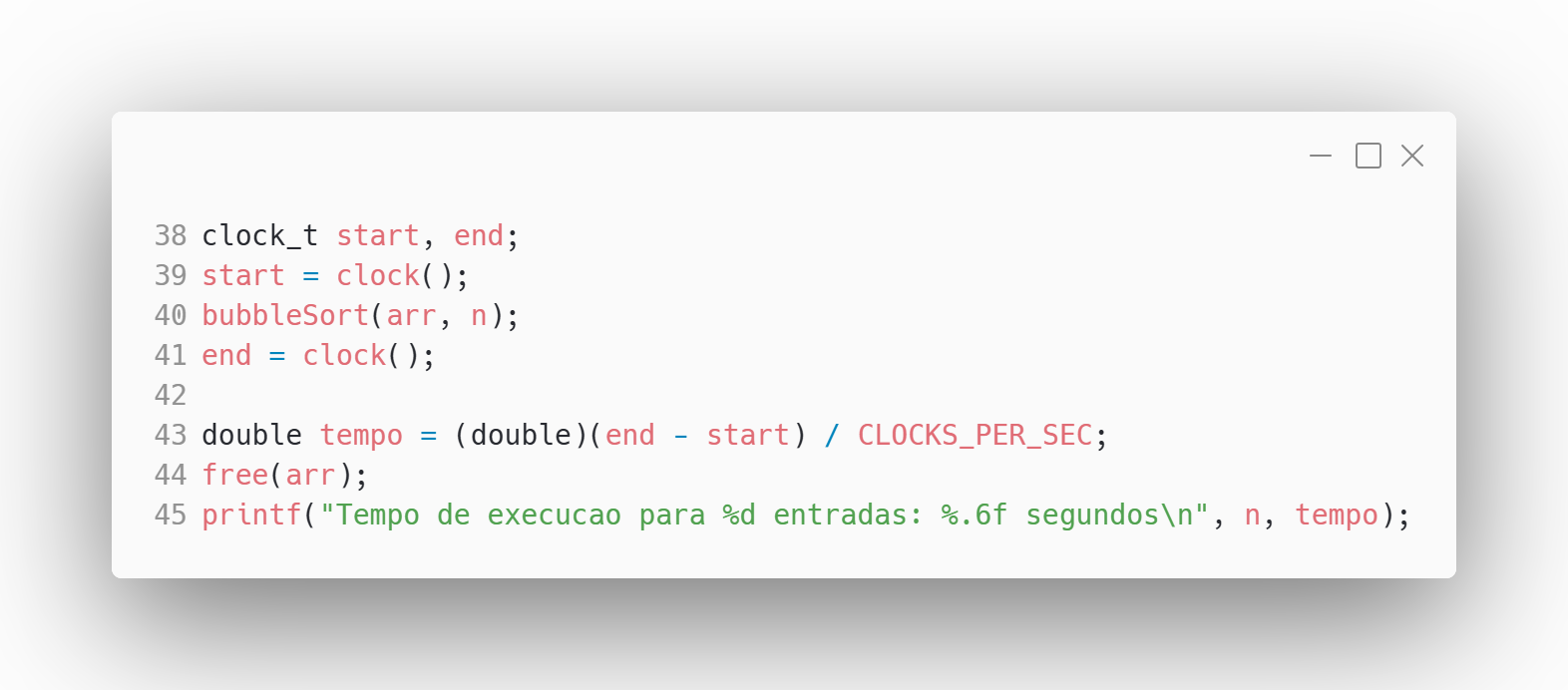
Da linha 21 a 23 está sendo definido o nome do arquivo a ser aberto. A variavel n recebe o

valor do tamanho do array atual, isto é, quando i = 1 ele vai buscar no array o indice 1 e vai colocar na variavel n para que posteriormente seja unida a variavel nome\_arquivo, que vai assim receber o valor de n e unir ao texto, tendo assim o nome do arquivo a ser aberto.

Na linha 24, o código tenta abrir o arquivo gerado pela linha anterior com a função fopen(). O arquivo é aberto no modo de leitura “r”. A função retorna um ponteiro file que aponta para o arquivo aberto. Se o arquivo não existir ou não puder ser aberto, o valor de file será NULL. Em seguida há uma verificação para garantir que o arquivo vou aberto corretamente. Se fopen() retornou NULL, o que significa que o arquivo não pode ser aberto, o códdigo entra nessa condição if. Uma mensagem de erro então é impressa.

Na linha 31, caso o arquivo tenha sido aberto corretamente, o código aloca dinamicamente memória para armazenar os números do arquivo. A função malloc() é usada para alocar n \* sizeof(int) bytes de memória. Isso cria um espaço para armazenar n inteiros, que depende do valor do array que está atualmente.

Da linha 32 a 36 está o funcionamento para passar o códoigo para o array, iniciando uma variável k para servir como posição no array e em um loop a função fscanf() é usada para ler os números inteiros do arquivo e o armazena na posição arr[k] do array até que chegue ao final do arquivo, indicado pela constante EOF (End Of File). Após isso o arquivo é fechado com a função fclose(), liberando os recursos do sistema que estavam sendo usados para abrir o arquivo.



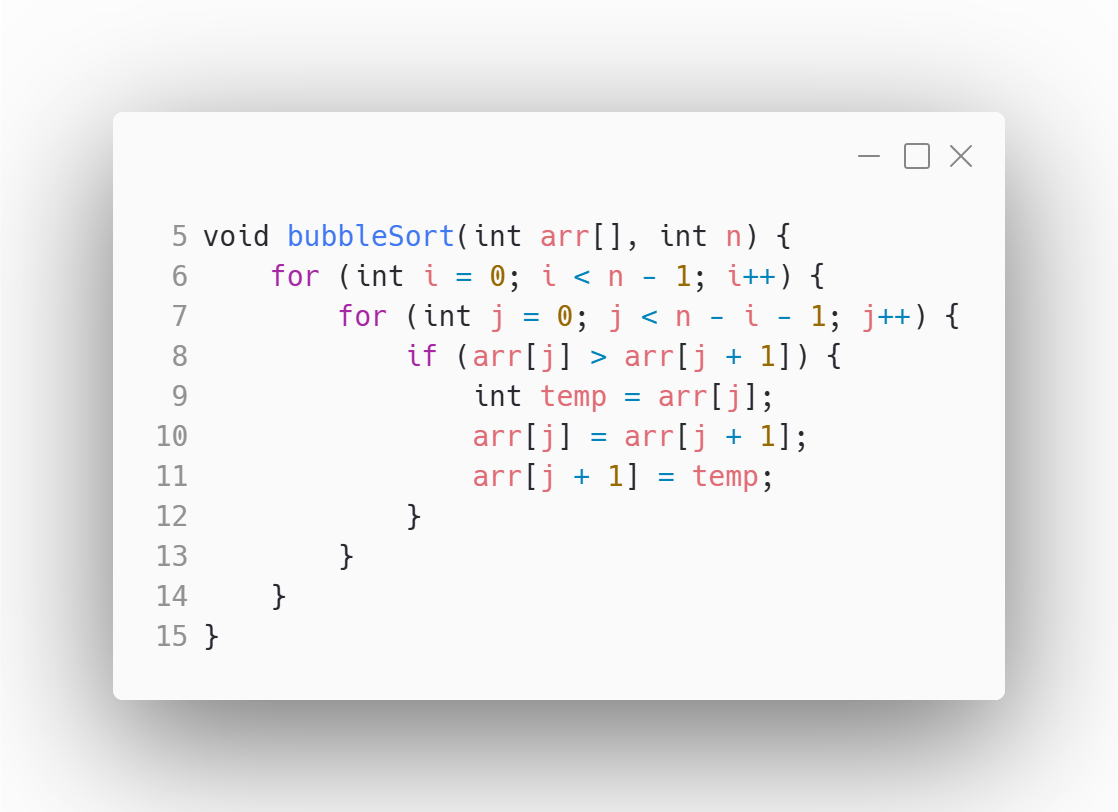
Na segunda parte do código principal é declado duas váriaveis, na linha 38, chamadas start e end, que serão usadas para armazenar o momento inicial e o momento final da execução do algoritmo de ordenação. A função clock(), na linha 39. é chamada para registrar o instante atual no qual a execução do código começa. Esse valor é armazenado na variável start.

O algoritmo de ordenação bubbleSort(arr, n) é executado neste ponto, onde arr é o array a ser ordenado e n é o número de elementos presentes no array. Esse é o código que será analisado quanto ao desempenho, ou seja, o tempo que leva para ordenar o array. Após a execução do algoritmo, a função clock() é chamada novamente para capturar o tempo final de execução,

armazenando o valor na variável end.

Na linha 43, a diferença entre o tempo final (end) e o tempo inicial (start) é calculada e convertida para segundos, dividindo o valor por CLOCKS\_PER\_SEC. Essa constante define quantos "clocks" correspondem a 1 segundo de tempo real. O valor resultante é armazenado na variável tempo, que representa o tempo de execução do algoritmo de ordenação em segundos.

Após o cálculo do tempo, a memória previamente alocada para o array arr é liberada com a função free(), para evitar vazamentos de memória. Por fim, o tempo de execução é impresso na tela utilizando a função printf(), informando o número de entradas e o tempo de execução calculado com uma precisão de 6 casas decimais.



A função bubbleSort é definida com dois parâmetros: arr[], um array de inteiros a ser ordenado, e n, o número de elementos no array. O objetivo desta função é ordenar o array de maneira crescente usando o algoritmo Bubble Sort.

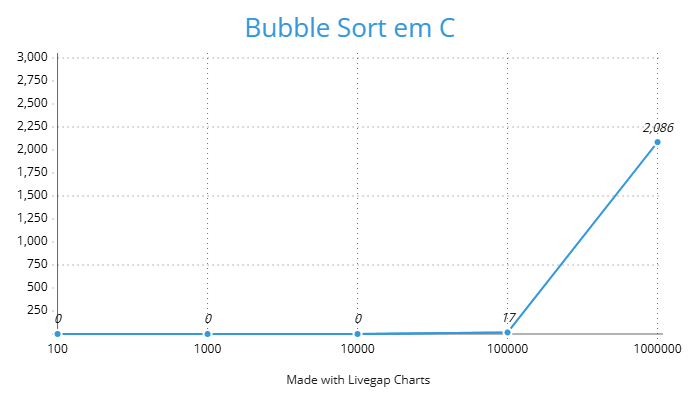
O primeiro loop (for (int i = 0; i < n - 1; i++)) controla o número de passagens feitas pelo array. Como a cada passagem o maior valor é encaminhado para o final do array, não é necessário continuar comparando o último elemento após cada passagem, o que justifica o uso de n - 1. Esse loop faz n - 1 iterações no total. O segundo loop (for (int j = 0; j < n - i - 1; j++)) percorre o array até a penúltima posição não ordenada a cada passagem. A variável i é usada para diminuir o número de comparações em cada passagem, uma vez que os elementos no final do

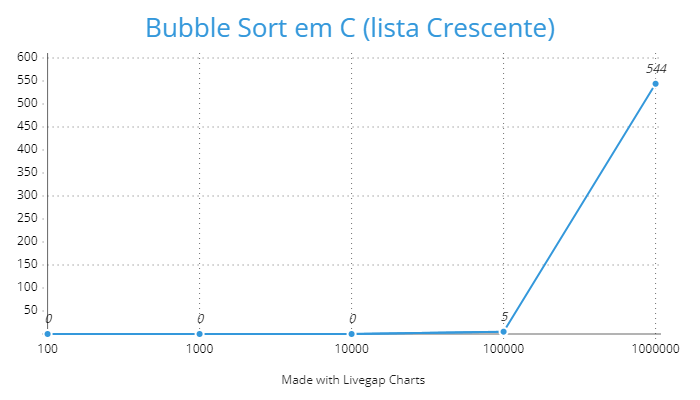
array já estarão ordenados após cada iteração do loop externo. Assim, a quantidade de comparações diminui à medida que a ordenação avança.

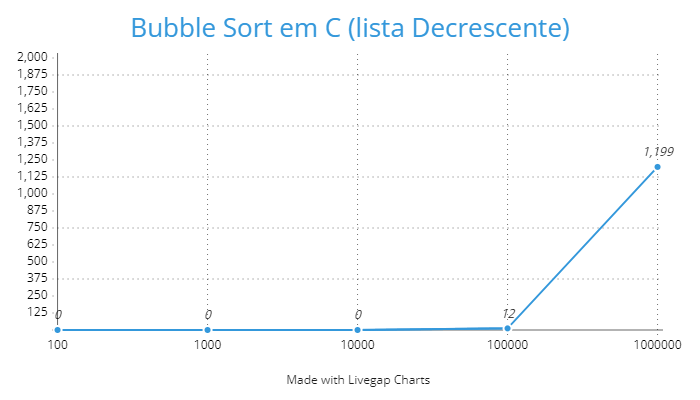
Dentro do loop interno, ocorre a comparação dos elementos adjacentes: arr[j] e arr[j + 1]. Se o elemento da posição j for maior do que o da posição j + 1, ocorre a troca entre eles. O processo de troca é feito utilizando uma variável temporária (temp), que armazena o valor de arr[j] temporariamente enquanto os valores de arr[j] e arr[j + 1] são trocados. O valor de arr[j] é substituído por arr[j + 1], e então o valor de arr[j + 1] é substituído pelo valor armazenado em temp, completando a troca.

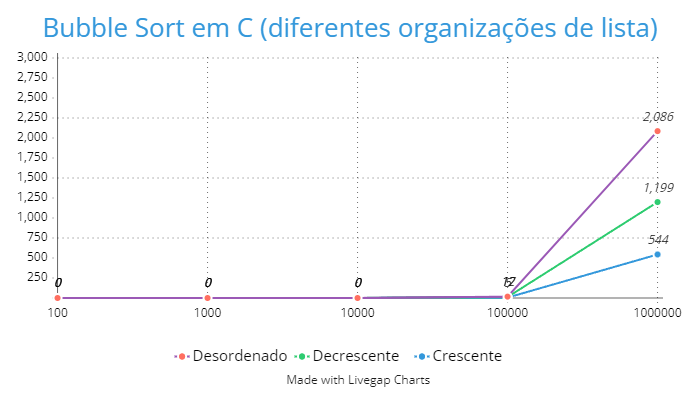
O algoritmo repete esse processo de comparação e troca para cada par de elementos adjacentes ao longo do array. Ao final de cada passagem do loop externo, o maior elemento não ordenado é colocado em sua posição correta no final do array, e assim, o algoritmo termina quando todas as passagens são concluídas, ou seja, quando o array está completamente ordenado.

**4.1.1.2. Gráficos**









**4.1.1.3. Análise**

Ao analisar o desempenho do algoritmo Bubble Sort implementado em C, observamos como o tempo de execução varia de acordo com diferentes tipos de listas de entrada: crescente, decrescente e desordenada. No caso da lista em ordem crescente, o algoritmo apresenta o melhor desempenho. Para entradas de até 10.000 elementos, o tempo de execução permanece em zero, o que indica que o algoritmo realiza poucas comparações e nenhuma troca. No entanto, com o aumento do tamanho da entrada, a execução se torna mais lenta, com 100.000 elementos levando 5 segundos e 1.000.000 de elementos atingindo impressionantes 544 segundos. Isso ocorre porque, embora o Bubble Sort seja eficiente para listas já ordenadas, o aumento no número de elementos causa um crescimento quadrático no tempo de execução.

Quando a lista está em ordem decrescente, o tempo de execução do Bubble Sort em C aumenta consideravelmente em relação à lista crescente. Para 100.000 elementos, o tempo sobe para 12 segundos, e para 1.000.000 de elementos, chega a 1.199 segundos. Isso acontece porque, em uma lista completamente inversa, o algoritmo precisa realizar o maior número possível de trocas a cada iteração, o que aumenta ainda mais a complexidade do processo. Apesar disso, a natureza eficiente da linguagem C ajuda a manter os tempos de execução moderados em relação ao que seria esperado de um algoritmo com complexidade O(n²).

No caso da lista desordenada, o Bubble Sort em C se comporta de forma ainda mais lenta. Embora o algoritmo tenha um tempo de execução muito baixo para entradas pequenas, como 100 e 1.000 elementos (com tempos de 0 e 0,001 segundos, respectivamente), ele começa a mostrar um desempenho muito ruim conforme o tamanho da entrada aumenta. Com 100.000 elementos, o tempo atinge 17,843 segundos, e para 1.000.000 de elementos, o tempo é de

impressionantes 2.086,794 segundos. Esse aumento significativo no tempo de execução reflete as limitações do Bubble Sort ao lidar com listas desordenadas, já que ele precisa realizar um grande número de comparações e trocas, e não possui qualquer mecanismo de otimização para entradas caóticas. Mesmo com a eficiência de C, o algoritmo não consegue compensar a complexidade do processo para listas grandes e desordenadas.

**4.1.2. Java**

**4.1.2.1. Implementação**



Na parte principal do código, na linha 18, a variável entradas[] é definida com 5 valores: 100, 1000, 10000, 100000 e 1000000. Esses valores representam os tamanhos dos arrays que o código vai processar. Para cada valor de entradas[i], será gerado um nome de arquivo específico e lido um conjunto de dados correspondente a esse tamanho de array.

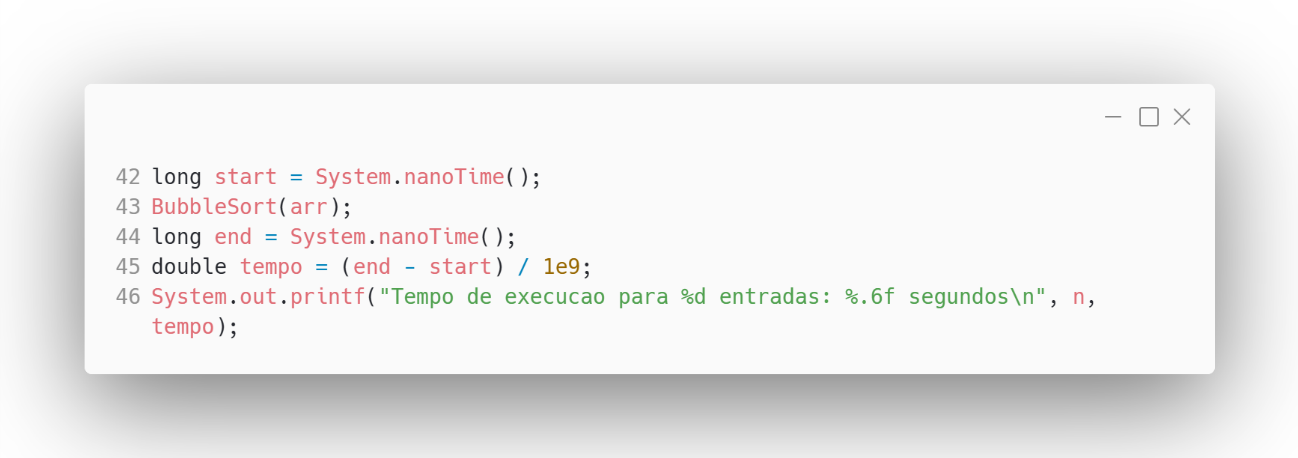
Na linha 20, é iniciado o loop externo, que será executado 4 vezes, no intuito de repetir os testes e assim possibilitar a medição de tempo para cada conjunto de dados. A intenção aqui é executar a mesma sequência de testes para calcular a média dos tempos de execução. Dentro do loop externo, há outro loop, que percorre o array entradas[], que tem 5 elementos (os tamanhos de entrada).

Na linha 22 e 23, o código define o nome do arquivo a ser aberto. A variável n recebe o valor do tamanho do array atual, isto é, quando i = 1, ela vai buscar o valor entradas[1] (1000) e colocá-lo na variável n. Posteriormente, o nome do arquivo é gerado pela concatenação de n com o texto "-entradas.txt", o que cria um nome de arquivo como 100-entradas.txt, 1000-entradas.txt, etc. O nome gerado será usado para localizar e abrir o arquivo correspondente ao tamanho do array.

Na linha 24, o código tenta abrir o arquivo gerado pela linha anterior com a função new File(nomeArquivo), que cria um objeto File apontando para o arquivo especificado. Em seguida, há uma verificação para garantir que o arquivo foi aberto corretamente. Se o arquivo não existir, o código exibe a mensagem de erro "Erro ao abrir o arquivo" e encerra a execução do programa com return.

Na linha 31 a 39, se o arquivo foi aberto corretamente, o código segue para a leitura do conteúdo do arquivo. O código usa um Scanner para ler os números inteiros do arquivo. Cada número lido é armazenado na lista numeros. Isso é feito até que todos os números do arquivo sejam lidos, utilizando o método hasNextInt() para garantir que apenas números inteiros sejam capturados. Caso ocorra algum erro durante a leitura será lançada uma exceção FileNotFoundException, e uma mensagem de erro será exibida no console.

Após a leitura dos números do arquivo, na linha 40, o código converte a lista numeros (que contém objetos Integer) para um array de inteiros arr[] utilizando o método stream() e a função mapToInt(i -> i). Isso cria um array de inteiros com os valores lidos do arquivo, pronto para ser processado no algoritmo de ordenação.



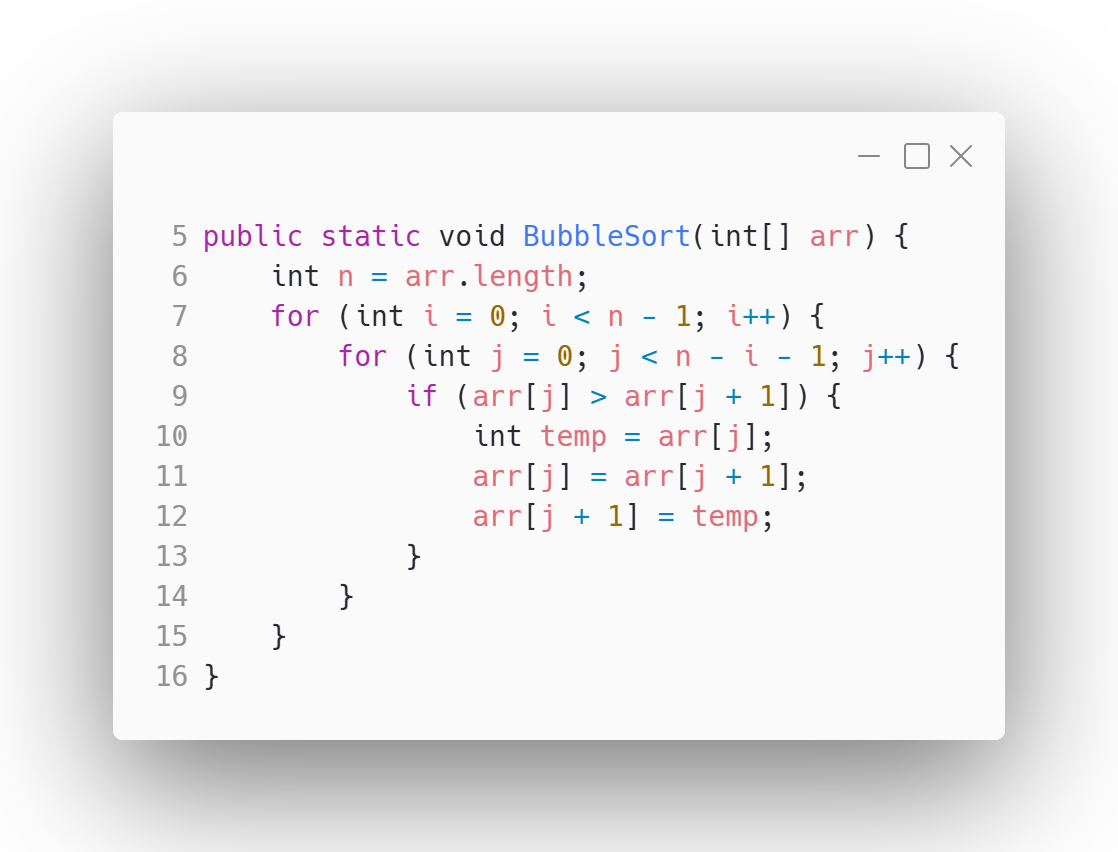
Na segunda parte do código principal, na linha 42 e 44, são declaradas duas variáveis chamadas start e end, que têm a finalidade de armazenar o momento inicial e o momento final da execução do algoritmo de ordenação. A função System.nanoTime() é chamada

para registrar o instante exato no qual o código começa a ser executado. Esse valor é armazenado na variável start e end.

O algoritmo de ordenação BubbleSort(arr) é executado entre as declarações das variaveis, na linha 43, onde arr é o array a ser ordenado. Esse é o trecho do código que será analisado em termos de desempenho, ou seja, o tempo que o algoritmo leva para ordenar o array.

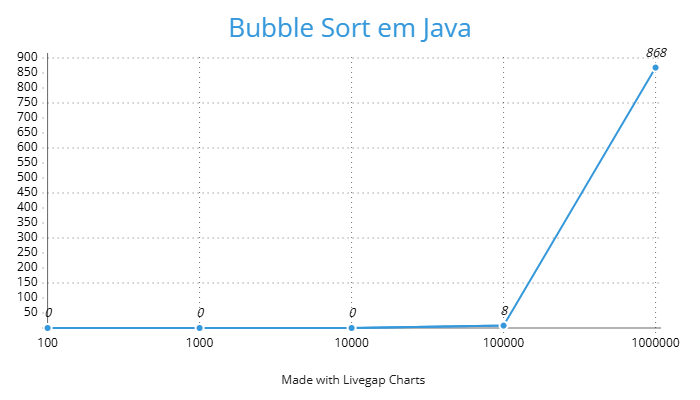
Na linha 45, a diferença entre o tempo final (end) e o tempo inicial (start) é calculada e, para convertê-la de nanossegundos para segundos, o valor é dividido por 1e9. Esse valor (1 bilhão) representa a quantidade de nanossegundos em um segundo. O resultado dessa operação é armazenado na variável tempo, que agora contém o tempo de execução do algoritmo de ordenação em segundos.

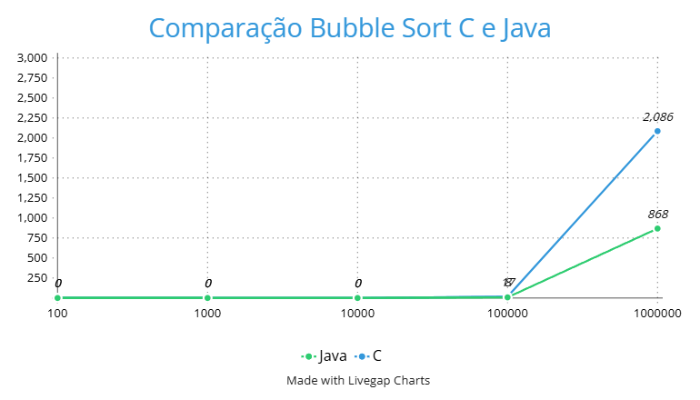
Por fim, na linha 46, o tempo de execução é impresso na tela usando o método System.out.printf(), que formata a saída para exibir o número de entradas e o tempo calculado com uma precisão de 6 casas decimais. Isso fornece ao usuário uma visão clara de quanto tempo foi necessário para ordenar o array de acordo com o número de elementos presentes nele.

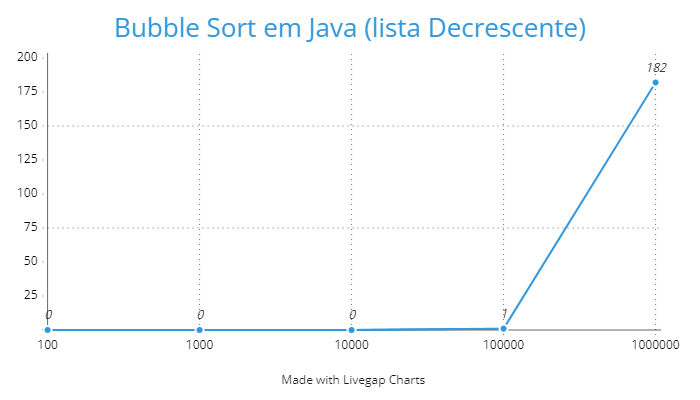


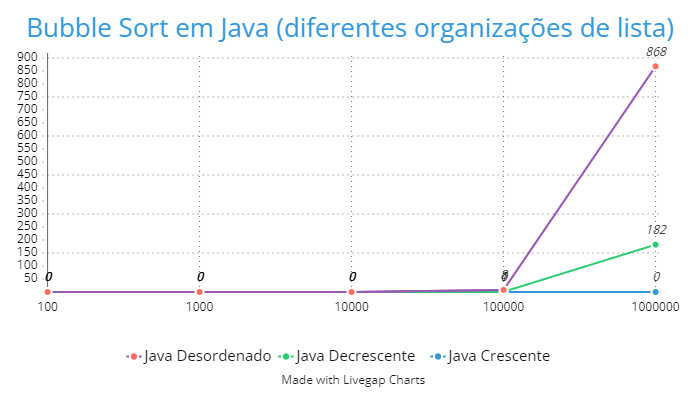
A função BubbleSort, com funcionamento semelhante ao visto anteriormente, está implementada de forma a receber como paramentro somente o array, sendo calculado o tamanho deste dentro da função via arr.lenght, na linha 6, que não existe em C, no qual exige uma manipulação mais manual.

**4.1.2.2. Gráficos**









**4.1.2.3. Análise**

Ao analisar o desempenho do algoritmo Bubble Sort implementado em Java, é possível observar que a linguagem apresenta um desempenho consideravelmente superior em comparação ao código em C, especialmente em listas desordenadas. Para a lista em ordem crescente, o algoritmo tem um desempenho muito bom, com tempos de execução praticamente nulos para todas as entradas, independentemente do tamanho. Mesmo para

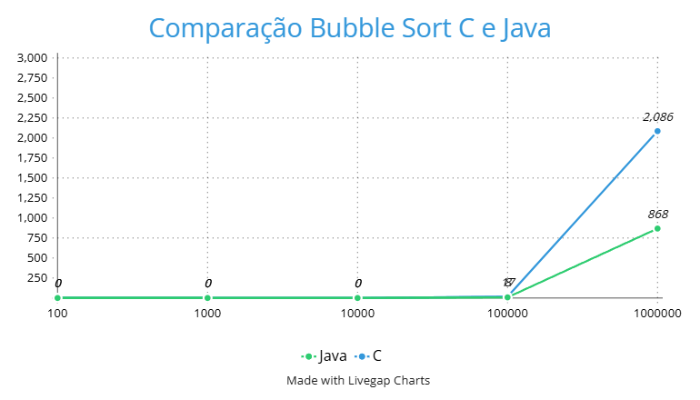
1.000.000 de elementos, o tempo permanece em zero segundos. Esse desempenho pode ser

explicado pelas otimizações dinâmicas da Máquina Virtual Java (JVM), que consegue lidar com listas ordenadas de forma eficiente, realizando otimizações em tempo de execução.

Quando a lista está em ordem decrescente, o tempo de execução em Java ainda é muito rápido, especialmente para entradas pequenas, com tempos de 0 segundos para 100 e 1.000 elementos. Para 100.000 elementos, o tempo é de apenas 1 segundo, e para 1.000.000 de elementos, o tempo atinge 182 segundos. Embora o Bubble Sort precise realizar o maior número de trocas para listas inversamente ordenadas, a JVM consegue otimizar a execução de forma que a diferença de desempenho em relação a C seja mais controlada, apesar do aumento no número de elementos.

O maior ganho de desempenho em Java é observado nas listas desordenadas. Para entradas pequenas, como 100 e 1.000 elementos, o tempo de execução é muito baixo (0,000046 e 0,001349 segundos, respectivamente), mas conforme o tamanho da entrada cresce, a diferença de desempenho se torna ainda mais notável. Para 10.000 elementos, o tempo é de 0,030079 segundos, e para 100.000 elementos, o tempo atinge 8,388 segundos. No entanto, o grande salto ocorre para 1.000.000 de elementos, quando o tempo de execução atinge 868,95 segundos. Mesmo assim, em todos os casos, o desempenho em Java continua muito mais eficiente do que em C, refletindo a capacidade da JVM de otimizar a execução durante o processo.

**4.1.3 Comparação**

****

Em conclusão, a análise do desempenho do algoritmo Bubble Sort nas linguagens C e Java revela diferenças significativas, principalmente no que diz respeito ao tempo de execução em

função do tipo e tamanho da entrada. Enquanto o Bubble Sort em C mostra uma performance mais eficiente para listas ordenadas, o tempo de execução aumenta consideravelmente quando as listas estão desordenadas, evidenciando a natureza ineficiente do algoritmo para esse tipo de entrada. O crescimento quadrático do tempo de execução em C é evidente, especialmente para entradas grandes, como demonstrado pelos tempos elevados para 1.000.000 de elementos.

Por outro lado, em Java, a situação é bastante diferente. A Máquina Virtual Java (JVM) oferece otimizações em tempo de execução que tornam o desempenho do Bubble Sort muito mais eficiente, particularmente para listas desordenadas. Mesmo que o algoritmo ainda tenha uma complexidade quadrática, a JVM consegue mitigar o impacto dessa complexidade com otimizações dinâmicas, o que resulta em tempos de execução mais baixos, mesmo para entradas grandes. Em listas crescentes, por exemplo, o tempo de execução permanece praticamente nulo para qualquer tamanho de entrada, algo que não ocorre em C. Nas listas decrescentes e desordenadas, o desempenho de Java também se mantém superior ao de C, com a diferença se tornando ainda mais notável à medida que o tamanho da entrada cresce.

Portanto, embora ambos os algoritmos apresentem limitações devido à complexidade do Bubble Sort, a linguagem Java, com a ajuda da JVM, consegue superar as limitações de C, proporcionando um desempenho significativamente melhor, especialmente para listas desordenadas. Essa análise reforça a importância de considerar o ambiente de execução ao escolher a linguagem para implementar algoritmos, já que a JVM em Java oferece vantagens substanciais em termos de otimização e eficiência, mesmo em algoritmos de complexidade elevada como o Bubble Sort.

**4.2. INSERTION SORT**

**4.2.1. C**

**4.2.1.1. Implementação**

****

Este novo código em C representa uma evolução significativa em relação ao código original, com uma organização mais modular e estruturada. No código anterior, a lógica estava concentrada em uma única função, o que limitava a flexibilidade e a manutenção do programa. Já neste novo código, o processo foi dividido em várias partes, cada uma com uma responsabilidade específica, o que torna o programa mais legível, modular e fácil de modificar.

Mantendo a mesma lógica para o número de entradas, dessa vez o número de elementos no array entradas[] é calculado dinamicamente através da expressão sizeof(entradas) / sizeof(entradas[0]). Isso garante que o programa possa se adaptar a qualquer alteração no número de entradas sem necessidade de modificações manuais no código.

O código então chama uma função chamada processarArquivosDuracao(entradas, numEntradas), que será visto a seguir.