UNIVERSIDADE PAULISTA

BACHAREL EM CIÊNCIA DA COMPUTAÇÃO

DIEGO FREIRE DE ALMEIDA, N8059D2

KAIKY DE LARA SALES, N9218H8

LEONARDO DE SOUZA RODRIGUES, F344HB2

NÍCOLAS PIMENTA DA SILVA, N863579

VITOR DOS SANTOS ROSA, G521CE3

ALGORITMOS DE ORDENAÇÃO DE DADOS: UMA ANÁLISE COMPARATIVA SOBRE ALGUMAS TÉCNICAS UTILIZADAS

Jundiaí – SP

2023

DIEGO FREIRE DE ALMEIDA, N8059D2

KAIKY DE LARA SALES, N9218H8

LEONARDO DE SOUZA RODRIGUES, F344HB2

NÍCOLAS PIMENTA DA SILVA, N863579

VITOR DOS SANTOS ROSA, G521CE3

ALGORITMOS DE ORDENAÇÃO DE DADOS: UMA ANÁLISE COMPARATIVA SOBRE ALGUMAS TÉCNICAS UTILIZADAS

Artigo das atividades práticas supervisionadas, apresentado ao Curso de Ciência da Computação para composição de nota.

Orientador: Prof. Sérgio Soares

Jundiaí – SP

2023

ÍNDICE

1. OBJETIVO DO TRABALHO

Este artigo visa realizar uma pesquisa quantitativa sobre o desempenho de quatro algoritmos de ordenação: *Bubble Sort*, *Shell Sort*, *Selection Sort* e *Quick Sort*. A análise compreenderá a avaliação comparativa de sua eficiência temporal e espacial, utilizando métricas específicas, como uso de memória, por exemplo. O estudo busca contribuir para a compreensão detalhada das características de cada algoritmo, fornecendo *insights* para a escolha informada do método mais adequado em diferentes cenários.

Assim, ao finalizar esta investigação, espera-se que os resultados obtidos forneçam uma base sólida para a compreensão comparativa do desempenho dos algoritmos de ordenação estudados, contribuindo para a eleição do ideal algoritmo, entre as quatros opções apresentadas, para a ordenação dos dados de um satélite capaz de gerar, armazenar e catalogar em torno de 100 mil imagens de determinada região a cada 24 horas. As imagens serão ordenadas por latitude, conforme o nome de cada imagem, que se refere a posição e coordenadas de onde a foto foi capturada.

2. INTRODUÇÃO

A ordenação de dados, é um conceito na ciência da computação utilizado para otimizar a organização de informações para facilitar a leitura e tratativa deles, um exemplo de tal utilização, pode ser de uma lista telefônica com diversos contatos, imagine ter que vasculhar toda a lista aleatoriamente até encontrar a informação desejada sem ter um indicativo de chegada ao objetivo, por isso a alternativa seria ordenar os dados, como por exemplo ordenar os nomes em ordem alfabética assim o usuário da lista pode ter uma referência de onde ele está e qual é a distância que ele está do objetivo, por isso se tratando de grandes bases de informação é essencial a ordenação de dados alinhada com os requisitos do projeto para assim ordenar com melhor otimização possível, pois mesmo ordenando os dados temos que nos atentar ao tipo de ordenação que vamos utilizar por conta que dependendo do volume de dados que vamos analisar e ordenar podemos ter certa lentidão desnecessária ao utilizar uma ordenação não otimizada para o volume desejado.

Alguns desses algoritmos que analisamos para esse projeto de ordenação de dados são:

- *Bubble Sort;*
- *Quick Sort;*
- *Selection Sort;*
- *Shell Sort.*

Na questão de hardware, para testar os algoritmos em cenários de desempenho maior e desempenho menor, utilizamos as seguintes máquinas:

Maior desempenho:

Sistema Operacional: Ubuntu 20.04

Processador: AMD EPYC 7763

Memória Ram: 16GB

Menor desempenho:

Sistema Operacional: Ubuntu 20.04

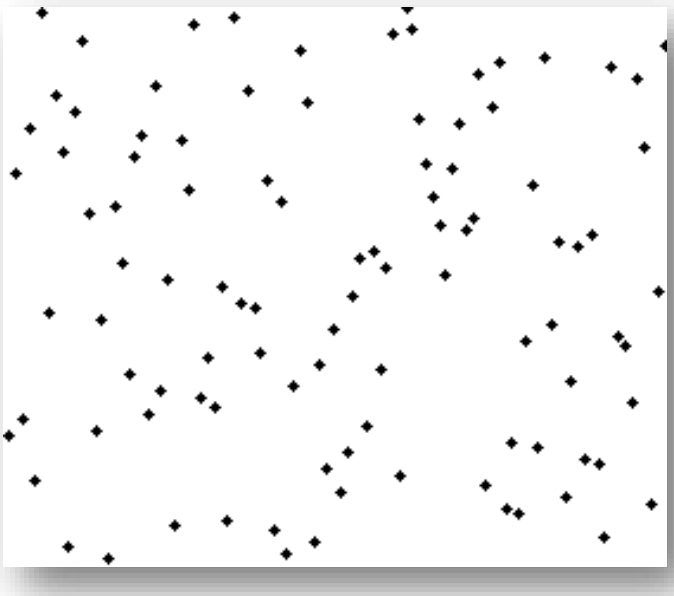
Processador: Intel Xeon E5-2673

Memória Ram: 1GB

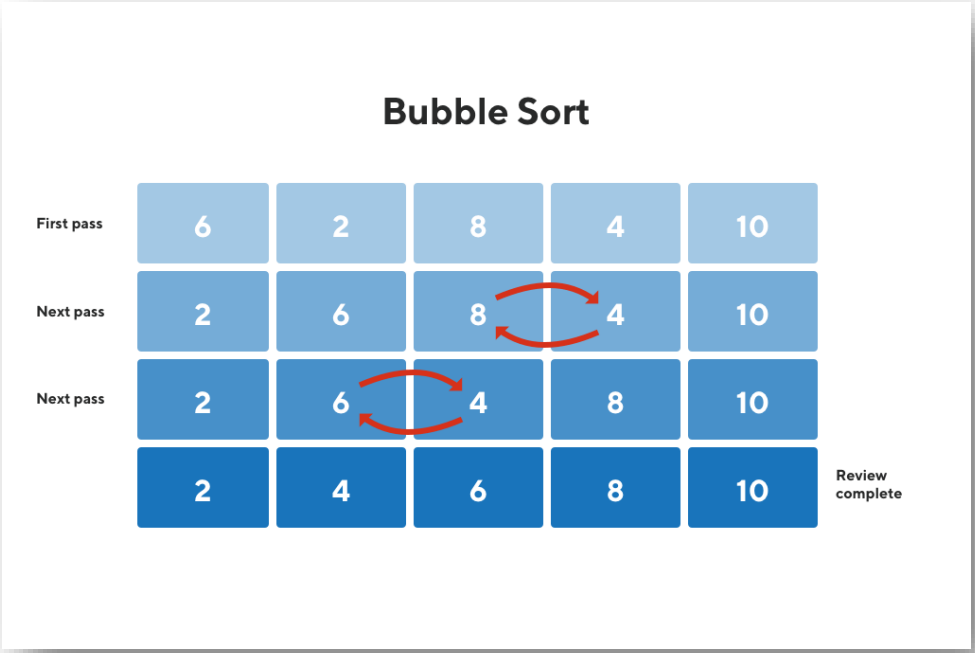
Utilizamos dois cenários para termos ciência de como os recursos da máquina iram afetar os algoritmos de ordenação e se algum terá vantagem se tiver mais recursos a disposição na hora da execução.

2.1. Bubble Sort

O algoritmo de classificação por bolha, ou apenas *Bubble Sort*, se trata de uma estrutura de ordenação de dados, que possui como objetivo final sequenciar os itens de um conjunto de dados em determinada ordem. Este método consiste em examinar cada os dados em pares, comparando um elemento a seu adjacente, verificando qual possui maior valor e em seguida trocando suas posições dentro da ordem estipulada, partindo do primeiro, de modo que o maior valor seja enviado para a última posição de maneira sumária (em uma ordenação crescente) “como uma bolha”, por isso, *Bubble Sort*.



https://pt.wikipedia.org/wiki/Bubble_sort



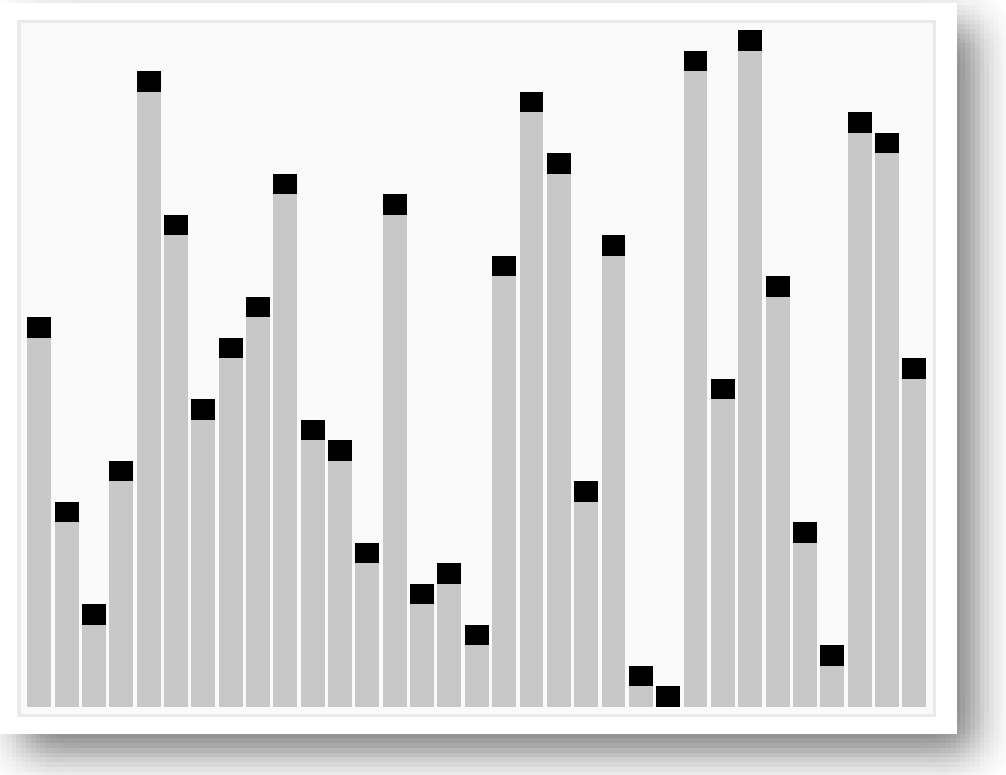
https://www.productplan.com/glossary/bubble-sort/

2.2. Quick Sort

O Algoritmo Quick Sort foi criado em 1960 por um cientista da computação britânico chamado Charles Antony Richard Hoare, quando estava tentando traduzir um dicionário de Inglês para Russo no desenvolvimento de um projeto de tradução de máquina na Universidade de Moscovo. A ideia de Charles era de dividir o problema em diversas pequenas etapas, a fim de facilitar a sua resolução com menos tempo e esforço, nascendo assim o algoritmo *Quicksort*, onde após diversos aprimoramentos, foi publicado em 1962. O *Quicksort* tem a ideia principal de dividir os dados em duas partes partindo de um único elemento presente dentro desses dados. Este elemento é denominado de pivô.

O pivô pode ser selecionado de quatro formas principais: o primeiro elemento dos dados, o segundo elemento dos dados, o elemento central dos dados – ou seja, o elemento que os divide no meio, uma seleção aleatória de qualquer elemento dentro dos dados

Ao definir o pivô, o algoritmo irá comparar este elemento em relação a todos os outros, colocando os elementos que forem menores que ele a sua esquerda, e os elementos que forem maiores, à sua direita. Esta operação é chamada de



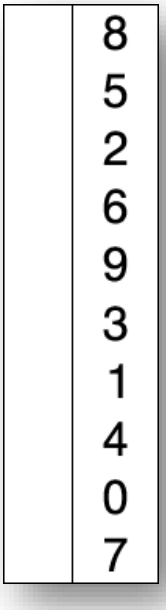
“particionamento”. Após o primeiro pivô ser selecionado e testado, o próximo pivô é

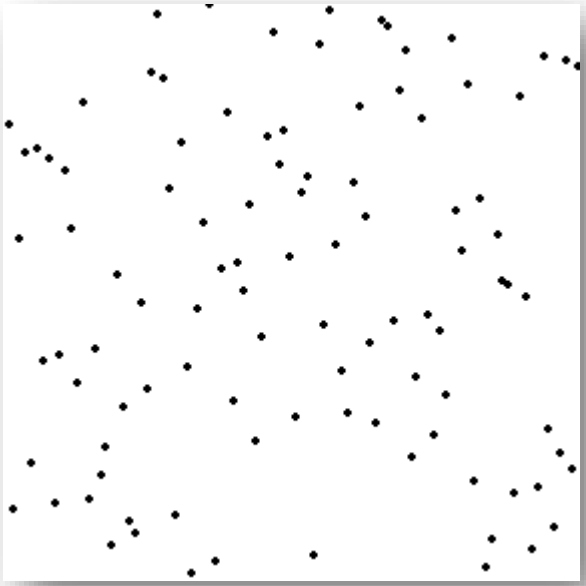
https://pt.wikipedia.org/wiki/Quicksort

eleito e então comparado com os elementos restantes e ainda não ordenados. No fim deste processo, a base de dados inteira estará ordenada.

2.3. Selection Sort

Esse algoritmo de ordenação não possui um criador definido ou uma de criação exata, ele é reconhecido como um algoritmo simples e de fácil implementação, dentre os algoritmos que utilizamos no projeto ele é o menos performático por conta da sua forma de funcionamento que segue a seguinte formula, o algoritmo percorre uma lista de dados até o fim procurando o menor valor nela a partir da primeira posição, assim que encontra esse valor ele move o mesmo para a primeira posição e o antigo valor da primeira posição para a posição que ele alcançou.





https://pt.wikipedia.org/wiki/Selection_sort

2.4. Shell Sort

O algoritmo de ordenação Shell Sort, desenvolvido por Donald Shell em 1959, é uma técnica de ordenação que busca aprimorar o método de ordenação por inserção. Assim como o Bubble Sort e o Quick Sort, o Shell Sort visa sequenciar elementos de uma lista em uma ordem específica. No entanto, ele adota uma abordagem diferente, introduzindo a ideia de ordenação por incremento. Ao contrário

do Bubble Sort, que compara elementos adjacentes, e do Quick Sort, que utiliza o conceito de pivô para dividir e conquistar, o Shell Sort começa ordenando elementos distantes uns dos outros, usando um intervalo chamado de "incremento". Esse incremento é gradualmente reduzido até atingir 1, momento em que a lista está quase totalmente ordenada. A última fase, geralmente realizada com um algoritmo de ordenação por inserção, ajusta os detalhes finais.

A escolha dos incrementos no Shell Sort é crucial para sua eficiência. Diferentes sequências de incremento podem ser utilizadas, como a sequência de Knuth (1, 4, 13, 40, ...) ou a sequência de Ciura, dependendo do contexto.

A característica fundamental do Shell Sort é a combinação de eficiência de algoritmos de ordenação por incremento com a simplicidade do algoritmo de ordenação por inserção. Embora não seja tão rápido quanto alguns algoritmos mais avançados em certas situações, o Shell Sort apresenta um desempenho geralmente melhor que o Bubble Sort e é mais fácil de implementar do que o Quick Sort em muitos casos. Isso o torna uma escolha viável em diversos contextos de ordenação de dados.

https://pt.wikipedia.org/wiki/Shell_sort https://simple2code.com/java-programs/shell-sort-in-java/

3. REFERENCIAL TEÓRICO

É fato que, quando se trata de afirmar taxativa e sistematicamente a superioridade de desempenho, após a implementação de um determinado modelo algoritmo de ordenação, sobre outra distinta, e ainda por cima, para um sistema tão particular, mas ainda sim consideravelmente soturno em certos méritos técnicos, que visa catalogar imagens vindas de um satélite, mirando apenas em incêndios florestais, nos cercamos de diversas questões importantes para considerar, como: o poder do equipamento utilizado pra rodar o sistema, a infraestrutura do ambiente onde os dispositivos estão posicionados, a necessidade (ou não) de escalabilidade da implementação, facilidade de manutenção no código fonte, usabilidade do produto final e muitos outras, portanto, subentendemos que, para que uma análise comparativa, como esta, seja congruente, dois importantes fatores são essenciais para balizar nosso trabalho neste artigo: metodologia e tecnologia. Dois itens que serão aqui explicados.

3.1. Metodologia

Conforme diz o Prof. Dr. Gerson Pastre de Oliveira (2019), “Uma pesquisa de caráter qualitativo é descritiva, sendo que palavras ou imagens são mais adequadas à descrição do que os número”(p. 20), portanto, estaremos aqui vocacionados em substanciar os dados de maneira ilustrativa e expositiva, baseando nossas discussões através da apresentação de gráficos e tabelas contextualizadas, apresentando a relação tempo(em segundos) e objetos(linhas lidas).

O foco aqui será todo o processo de pesquisa e implementação, muito mais do que os dados e conclusões realizadas a partir da abstração do trabalho de levantamento de dados.

Antes de toda a exposição desses procedimentos de arquitetura e criação desse sistema de ordenação, é válido realizar um estudo técnico detalhado de cada algoritmo utilizado nesta implementação. Estes, anteriormente mencionados e tendo seus respectivos caráteres conceituais e originários brevemente explanados, serão a seguir detalhados em seus méritos técnicos e práticos prevendo compreender, através de ilustrações claras e objetivos, o funcionamento de cada um deles.

3.2. Ordenando dados em bolhas

O algoritmo de ordenação *Bubble sort*, também chamado de *jump-down sort*, é reconhecido por sua simplicidade, mas sua eficiência é limitada, sendo mais adequado para conjuntos de dados pequenos. A seguir, apresentamos uma descrição técnica passo-a-passo do seu funcionamento.

Comparação de Pares Adjacentes:

- O algoritmo inicia comparando o primeiro elemento com o segundo na lista.
- Se o primeiro elemento for maior que o segundo, ocorre uma troca. Caso contrário, os elementos permanecem inalterados.

Iteração pela Lista:
- O processo de comparação e possível troca é então repetido para o segundo e terceiro elemento, e assim por diante, até o penúltimo par de elementos na lista.

Deslocamento Iterativo:
- A cada iteração completa pela lista, o maior elemento entre os não ordenados é movido para a posição correta no final da lista.
- O processo é repetido até que todos os elementos estejam ordenados.

Iterações Subsequentes:
- O algoritmo continua realizando essas iterações até que nenhuma troca seja necessária, indicando que a lista está ordenada.

Exemplo em C:

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // Comparação e troca se necessário
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

    printf("Array ordenado: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Conforme conclui Owen Astrachan (2003. p3), embora simples de entender e implementar, o algorítmo apresenta uma complexidade de tempo de $O(n^2)$, tornando-o menos eficiente para conjuntos de dados extensos. Sua utilização é mais apropriada em cenários onde a simplicidade do algoritmo supera suas limitações de desempenho.

Entender o *Bubble Sort* proporciona uma base valiosa para compreender algoritmos mais sofisticados e eficientes.

3.3. Ordenação rápida, mas não o suficiente

O algoritmo de ordenação *Quick Sort* é um método eficiente e amplamente utilizado para ordenar conjuntos de dados. Ele pertence à categoria de algoritmos de ordenação por comparação e segue a abordagem "dividir para conquistar". Aqui estão os detalhes técnicos de como o *Quick Sort* funciona:

Escolha do Pivô:

- O *Quick Sort* seleciona um elemento da lista como pivô. A escolha do pivô pode afetar o desempenho do algoritmo, e várias estratégias podem ser adotadas, como escolher o primeiro elemento, o último elemento ou um elemento aleatório.

Particionamento:

- A lista é reorganizada de modo que os elementos menores que o pivô, fiquem à esquerda, e os elementos maiores à direita. Isso é feito por meio de um processo de partição.

Recursividade:

- O *Quick Sort* é aplicado recursivamente às sublistas à esquerda e à direita do pivô. Cada sublista é então particionada e ordenada separadamente.

Base da Recursão:

- O processo recursivo continua até que as sublistas atinjam o tamanho de zero ou um, momento em que são consideradas ordenadas por definição.

Exemplo em C:

```c
#include <stdio.h>

// Função para trocar dois elementos
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Função para encontrar a posição correta do pivô na lista
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Escolha do pivô, pode ser ajustada
    int i = (low - 1);  // Índice do menor elemento
```

```
    for (int j = low; j <= high - 1; j++) {
        // Se o elemento atual for menor que ou igual ao pivô
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Função principal Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Encontre a posição correta do pivô na lista
        int pi = partition(arr, low, high);

        // Aplicar Quick Sort recursivamente às sublistas à esquerda e à direita do pivô
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Função para imprimir um array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Exemplo de uso
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    printf("Array ordenado: \n");
    printArray(arr, n);
    return 0;
}
```

3.4. Ordenação por seleção

O algoritmo de ordenação *Selection Sort* é um método simples e intuitivo para ordenar conjuntos de dados. Ele segue a abordagem de seleção repetida dos elementos mínimos (ou máximos) da lista e os posiciona na ordem desejada. Abaixo, apresento os detalhes técnicos de como o *Selection Sort* funciona:

Seleção do Elemento Mínimo:

- O *Selection Sort* inicia selecionando o elemento mínimo da lista.

Troca com o Primeiro Elemento:

- O elemento mínimo é trocado com o primeiro elemento da lista, colocando-o em sua posição correta.

Repetição para a Sublista Restante:

- O mesmo processo é então aplicado à sublista restante (excluindo o primeiro elemento), onde o próximo menor elemento é identificado e trocado com o segundo elemento da lista.

Iteração Completa:

- Esse processo de seleção e troca é repetido até que toda a lista esteja ordenada.

Exemplo em C:

```c
#include <stdio.h>
// Função para trocar dois elementos
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Função principal Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_index;

    // Percorre toda a lista
    for (i = 0; i < n - 1; i++) {
        // Encontra o índice do elemento mínimo na sublista não ordenada
        min_index = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index])
                min_index = j;
        }

        // Troca o elemento mínimo com o primeiro elemento não ordenado
        swap(&arr[min_index], &arr[i]);
    }
}

// Função para imprimir um array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Exemplo de uso
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);
```

```
    printf("Array ordenado: \n");
    printArray(arr, n);
    return 0;
}
```

3.5. Versátil como uma concha

O algoritmo de ordenação *Shell Sort* é uma variação do algoritmo de ordenação por inserção que visa melhorar o desempenho deste último em conjuntos de dados maiores. A abordagem principal do *Shell Sort* é comparar e trocar elementos distantes em vez de adjacentes, diminuindo gradualmente a distância entre os elementos a serem comparados. A seguir, apresento os detalhes técnicos de como o *Shell Sort* funciona:

Escolha dos Gaps (Intervalos):

- O Shell Sort começa escolhendo um conjunto de intervalos, também chamados de "gaps", que representam as distâncias entre os elementos a serem comparados. A escolha desses intervalos influencia diretamente o desempenho do algoritmo.

Aplicação do Algoritmo de Inserção:

- Para cada intervalo, o algoritmo aplica uma versão modificada do algoritmo de ordenação por inserção. Elementos que estão a uma distância de intervalo uns dos outros são comparados e trocados conforme necessário.

Redução Gradual dos Intervalos:

- O processo de ordenação é repetido com intervalos progressivamente menores, reduzindo gradualmente a distância entre os elementos a serem comparados.

Iteração Completa:

- O algoritmo continua reduzindo os intervalos até atingir um intervalo de 1. Nesse ponto, o Shell Sort executa uma última iteração usando o algoritmo de inserção padrão para garantir a ordenação final.

Exemplo em C:

```
#include <stdio.h>

// Função principal Shell Sort
void shellSort(int arr[], int n) {
    // Escolha dos intervalos (gaps)
```

```c
    for (int gap = n / 2; gap > 0; gap /= 2) {
        // Aplicação do algoritmo de inserção com intervalo gap
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;

            // Comparação e troca dos elementos com intervalo gap
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }

            arr[j] = temp;
        }
    }
}

// Função para imprimir um array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Exemplo de uso
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    shellSort(arr, n);

    printf("Array ordenado: \n");
    printArray(arr, n);
    return 0;
}
```

4. DESENVOLVIMENTO

Chegando aos finalmentes, tiraremos agora um espaço para comentar sobre a idealização do projeto como um todo, os algoritmos que quase foram implementados, sobre os processos de desenvolvimento do sistema. Aqui teremos uma visão aprofundada do processo de implementação, criação e desenvolvimento do sistema voltado para a análise do desempenho de quatro algoritmos de ordenação de dados. Neste estágio, nosso foco é detalhar a arquitetura do sistema, a metodologia de implementação dos algoritmos selecionados, assim como as estratégias adotadas para a coleta de dados e exportação externa das análises realizadas.

Ao abordar cada aspecto do desenvolvimento, buscamos oferecer uma compreensão clara do ambiente experimental, das escolhas de design e das considerações técnicas que permearam o desenvolvimento do sistema. Desta forma, esta seção serve como um guia completo para o entendimento da estrutura e funcionamento do nosso ambiente de análise, proporcionando aos leitores uma base sólida para a avaliação dos resultados apresentados posteriormente no artigo.

4.1. Idealização do projeto

Durante o processo de idealização, muitas ideias para a implementação foram cogitadas, tanto pela diversidade de possibilidades para solução. A concepção do presente projeto fundamenta-se na premissa de que o geoprocessamento de imagens da Floresta Amazônica constitui uma ferramenta essencial para a fiscalização de atividades relacionadas a crimes ambientais. Dada a vastidão da região, o emprego de tecnologias de sensoriamento remoto, em particular, a captação de imagens por satélites, emerge como uma abordagem promissora para monitorar e avaliar alterações no ambiente florestal em larga escala.

A Floresta Amazônica, com sua diversidade e extensão territorial, apresenta desafios singulares em termos de fiscalização ambiental. Os satélites orbitais disponíveis geram aproximadamente 100 mil imagens da região a cada 24 horas. Este volume substancial de dados requer uma abordagem meticulosa na organização, catalogação e interpretação das informações coletadas.

Algumas questões levantadas seguida das propostas de intervenção:

O que ordenar:

- Notamos o desafio de decidir qual atributo utilizar como parâmetro de ordenação, já que arquivos de imagem possuem uma lista incontável de metadados e atributos essenciais que podem ser considerados para a ordenação. Entretanto, a ordenação através da latitude, presente no nome de cada arquivo de imagem previamente definido, tornou a métrica utilizada na implementação.

Gestão de Dados em Escala Massiva:

- A coleta diária de um grande número de imagens impõe desafios significativos na gestão de dados. Estratégias eficientes de armazenamento e catalogação tornam-se cruciais para assegurar a acessibilidade e integridade dos dados ao longo do tempo. Solucionado com a utilização de soluções de armazenamento em nuvem e tecnologias distribuídas para facilitar o gerenciamento eficiente de grandes volumes de dados. Estratégias como o uso de bancos de dados escaláveis e sistemas de arquivos distribuídos otimizam a recuperação e o acesso aos dados.

Processamento de Imagens e Análise de Padrões:

- O processamento e análise de padrões em um conjunto tão extenso de imagens requerem algoritmos robustos e eficientes. A identificação de atividades suspeitas e potenciais crimes ambientais demanda métodos avançados de geoprocessamento e aprendizado de máquina. Que pode ser compreendido com a implementação de algoritmos de aprendizado de máquina avançados para o processamento automático de imagens e análise de padrões. Técnicas como redes neurais convolucionais podem ser exploradas para a identificação automatizada de padrões associados a atividades suspeitas.

Integração de Tecnologias:

- A integração eficaz de diferentes tecnologias, como sistemas de informação geográfica (SIG) e algoritmos de análise espacial, para extrair informações úteis a partir das imagens, é um desafio complexo que envolve a harmonização de diversas ferramentas e plataformas. E com esforços da equipe, entendemos que, o estabelecimento de padrões de integração e interoperabilidade entre diferentes tecnologias, como sistemas de informação geográfica (SIG) e algoritmos de análise espacial são muito versáteis na solução dessa questão. O uso de padrões

abertos e APIs (Interfaces de Programação de Aplicações) pode facilitar a interconexão eficiente entre diversas ferramentas e plataformas.

Aspectos Éticos e Legais:

- A utilização de tecnologias de geoprocessamento para fins de fiscalização requer uma consideração cuidadosa dos aspectos éticos e legais associados. A coleta e interpretação de dados de vigilância exigem conformidade estrita com normativas e legislações pertinentes. Que pode ser ultrapassada, com o oferecimento de treinamento especializado para profissionais envolvidos no projeto, abordando aspectos técnicos e éticos do geoprocessamento. a conscientização sobre as implicações éticas e legais é fundamental para garantir práticas responsáveis durante a execução do projeto.

É com base em todas essas questões que optamos por um sistema simples, que possui como objetivo fornecer ao usuário o relatório do tempo utilizado para ordenação de dados previamente definidos (desordenados e semiordenados), de acordo com a imputação de certos parâmetros da parte do usuário final.

4.2. Obtenção dos dados para ordenação

A fundamentação do presente estudo demanda a obtenção criteriosa de dados provenientes de fontes confiáveis e abrangentes. Para a análise do desempenho de quatro algoritmos de ordenação de dados no contexto de geoprocessamento da Floresta Amazônica, a principal fonte de informações é derivada de imagens de satélite, capturadas a partir de observatórios orbitais. A plataforma selecionada para a obtenção desses dados foi o "*Kaggle*", um serviço online que disponibiliza *datasets* forma gratuita na internet.

Procedimento de aquisição:

Seleção da Plataforma:

- A escolha do *dataset no Kaggle* baseou-se na sua reputação consolidada como um repositório confiável e acessível de datasets de diversas natureza. Esta plataforma oferece uma gama abrangente de dados, incluindo o dataset que utilizamos, sobre incêndios florestais. O dataset conta com imagens de alta

resolução capturadas por satélites em órbita, imagens essas adequadas para os objetivos do nosso estudo.

Download de Imagens:

- Os procedimentos de download foram conduzidos de acordo com as especificações temporais necessárias para o estudo. Foi realizado um processo de seleção criterioso, considerando a frequência de captura de imagens pelos satélites disponíveis na plataforma.

Armazenamento Local:

- Os conjuntos de imagens adquiridos foram armazenados localmente em pastas designadas no diretório raiz onde o sistema de análise está implantado. Essa organização facilita o acesso e a manipulação dos dados durante as fases subsequentes do projeto. Todas as imagens têm suas latitudes interpretadas e extraída para um arquivo “txt” que será ordenado pelo algoritmo escolhido pelo usuário.

Estrutura de Diretórios:

/aps4-main
/wildfire-tests
wildfire-10.txt
wildfire-100.txt
wildfire-500.txt
wildfire-1000.txt
wildfire-10000.txt
wildfire-30000.txt

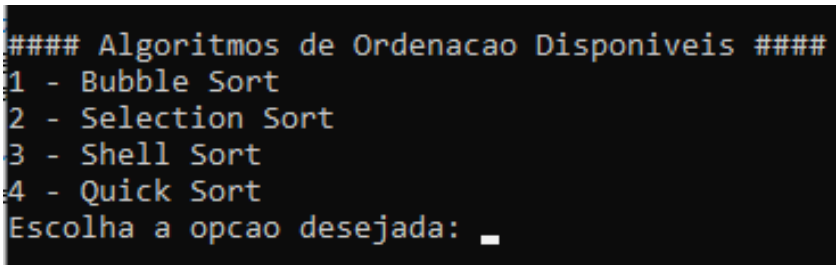
A estrutura de diretórios adotada organiza os dados em subpastas específicas, proporcionando uma hierarquia clara e facilitando a referência precisa durante a implementação e análise subsequente.

Essa fase de obtenção de dados constitui a base essencial para o desenvolvimento da análise de desempenho dos algoritmos de ordenação, uma vez que a qualidade e a representatividade dos dados são fatores cruciais para resultados robustos e conclusivos. O uso do EOS Data Analytics enriquece o projeto com dados

de satélite atualizados e relevantes para a região da Floresta Amazônica, permitindo uma avaliação mais precisa dos algoritmos sob investigação.

4.3. Processo de ordenação

O processo de ordenação se consiste na seguinte maneira: primeiro, o usuário seleciona qual algoritmo de ordenação quer utilizar selecionando o número correspondente a opção no índice:



Em seguida, ocorrem as interações com a respectiva função dentro do código fonte, conforme a opção selecionada.

1 – Bubble sort

```
void bubbleSort(char **arr, int n)
```

Esta, implementa o algoritmo *bubble sort* para ordenar os textos. O Bubble Sort compara pares adjacentes de strings e os troca se estiverem fora de ordem. Esse processo é repetido até que nenhuma troca seja necessária, garantindo que a string mais "pesada" (maior valor) seja movida para o final do vetor em cada passagem.

2 – Selection sort

```
void selectionSort(char **arr, int n)
```

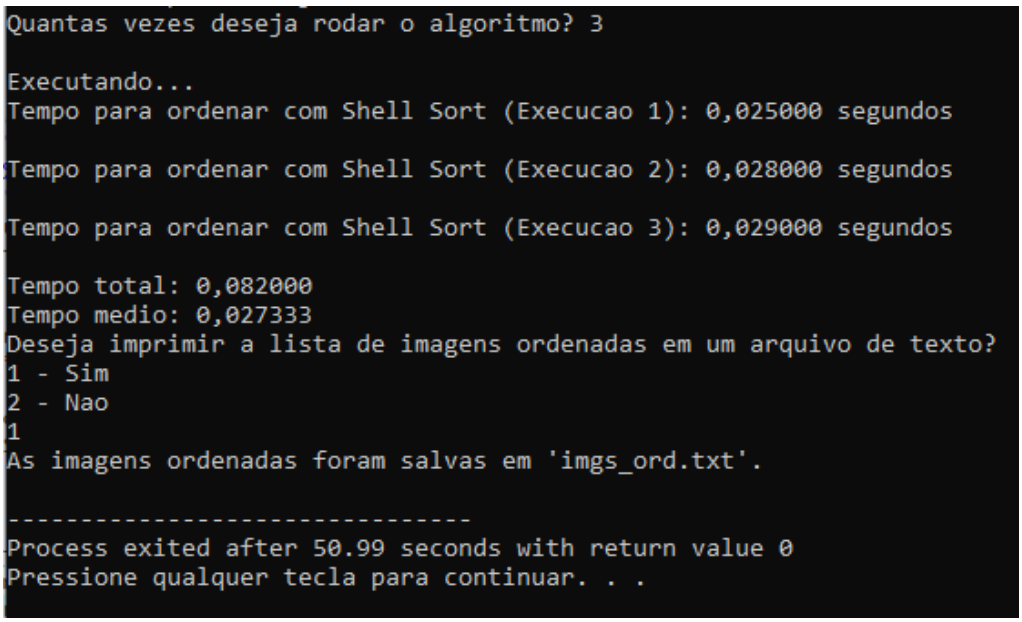
Esta implementa o algoritmo *selection sort.* O Selection Sort encontra iterativamente o menor elemento restante e o coloca na posição correta. Para fazer isso, o algoritmo percorre o vetor, identifica o menor elemento e o troca com o primeiro elemento não ordenado. Esse processo é repetido até que todo o vetor esteja ordenado.

3 – Shell sort

```
void shellSort(char **arr, int n)
```

Esta implementa o algoritmo *quick sort*. O Quick Sort utiliza a estratégia de dividir para conquistar. A função partition seleciona um pivô e rearranja os elementos ao redor dele, colocando os menores à esquerda e os maiores à direita. Essa operação é realizada recursivamente nas sub-listas resultantes até que todo o vetor esteja ordenado.

Em seguida, o usuário define quantas vezes o algoritmo vai rodar:



O sistema imprime cada execução e o tempo levado para ordenação em cada uma delas. Por fim, há ainda a possibilidade de salvar as imagens ordenadas no arquivo de texto ‘imgs_ord.txt’.

5. RESULTADOS E DISCUSSÃO

A seção de resultados e discussão apresenta uma análise abrangente da eficiência dos algoritmos de ordenação implementados em diferentes cenários de teste. Neste estudo, realizamos uma série de experimentos para avaliar o desempenho dos algoritmos (*Bubble Sort, Selection Sort, Shell Sort,* e *Quick Sort*) diante de variados tamanhos de conjuntos de dados e diferentes configurações, como dados ordenados, semi-ordenados e aleatórios.

O objetivo primordial desses testes é proporcionar uma compreensão aprofundada sobre como cada algoritmo responde a condições diversas, permitindo-nos identificar as vantagens e desvantagens de cada técnica em cenários específicos. A metodologia de avaliação abrange desde conjuntos de dados pequenos até

grandes, visando fornecer uma visão abrangente do desempenho em diferentes escalas.

A apresentação dos resultados é realizada por meio de tabelas e gráficos, proporcionando uma visualização clara e concisa das métricas de desempenho, como tempo de execução. Essas representações gráficas auxiliam na identificação de padrões de comportamento e na comparação direta entre os algoritmos.

Ao longo desta seção, os resultados obtidos são minuciosamente discutidos, destacando as tendências observadas e evidenciando as características distintivas de cada algoritmo em situações específicas. A discussão visa oferecer insights sobre as circunstâncias em que cada algoritmo se destaca, bem como possíveis limitações identificadas durante os experimentos.

Através dessa análise detalhada, almejamos fornecer uma base sólida para a compreensão das capacidades e limitações de cada algoritmo de ordenação, contribuindo assim para a tomada de decisões informadas sobre a escolha da técnica mais apropriada em diferentes contextos de aplicação.

5.1. Bateria de testes com dados semiordenados

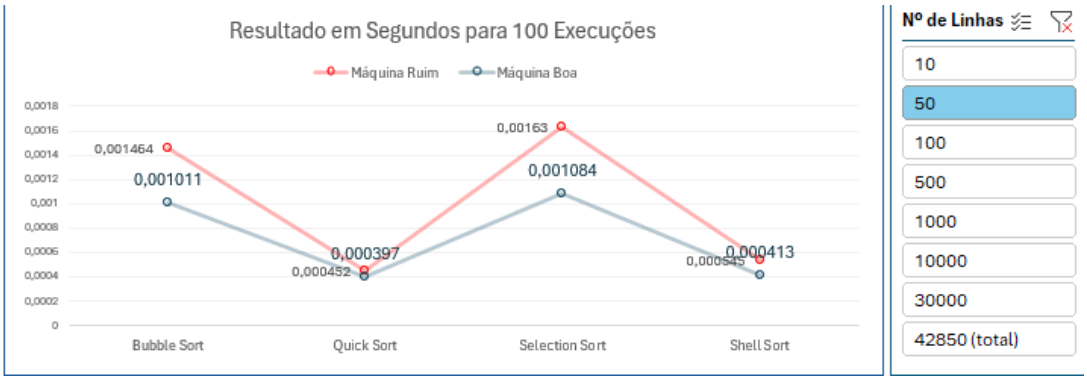
Nossa primeira bateria de testes se inicia com os dados sendo imputados de maneira semiordenada.

5.1.1. Tempo total

No que quesito: tempo total exigido para ordenação (em segundos). Podemos notar que, quanto ao bubble sort, existe uma visível e larga distância entre o desempenho total da “máquina ruim” com a denominada “máquina boa”, logo de início e com uma baixa quantidade no quesito tempo, o que denota bom desempenho,

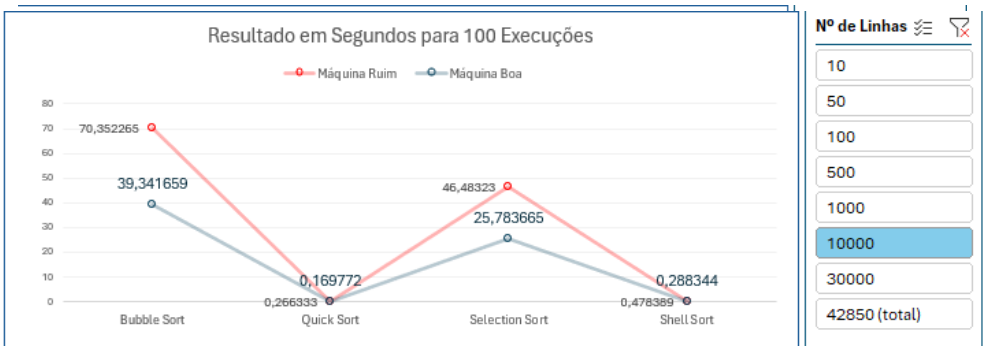
com 10 linhas. No quick sort, a diferença entre uma e outra não se dá de maneira tão significativa, enquanto entre selection e shell sort, existe uma gritante diferença se comparado com os demais, com exceção do quick sort, que mantém excelente desempenho em ambas as máquinas.

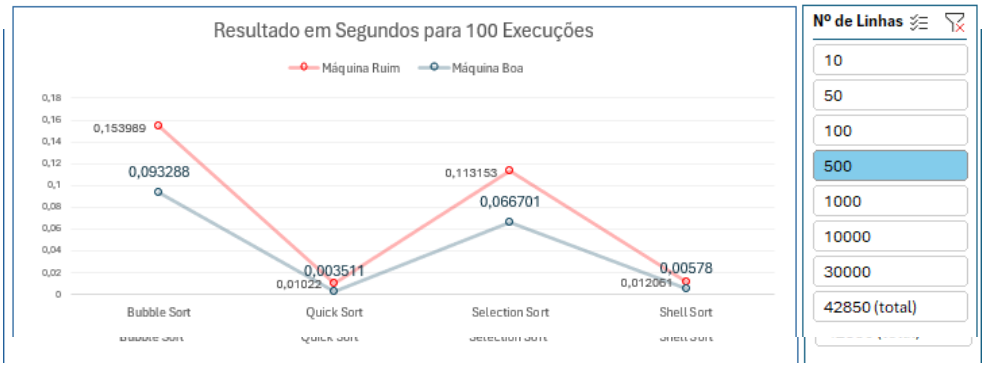
Ao aumentarmos o número de linhas para 50, notamos uma mudança gritante no desempenho do quick sort e do shell sort com os demais algoritmos. Aqui, o bubble sort, já começa a dar seus primeiros de defasagem em comparação com seus pares.



Com 100 linhas, o bubble sort acaba mostrando pior desempenho no tempo acumulado. Selection sort tem relativa melhora, e quick sort e shell sort seguem isolados como os mais velozes. Aqui as duas máquinas se aproximam no shell sort.

A partir das 500 linhas, temos a configuração de um padrão, onde os 4 algoritmos mantêm suas posições.

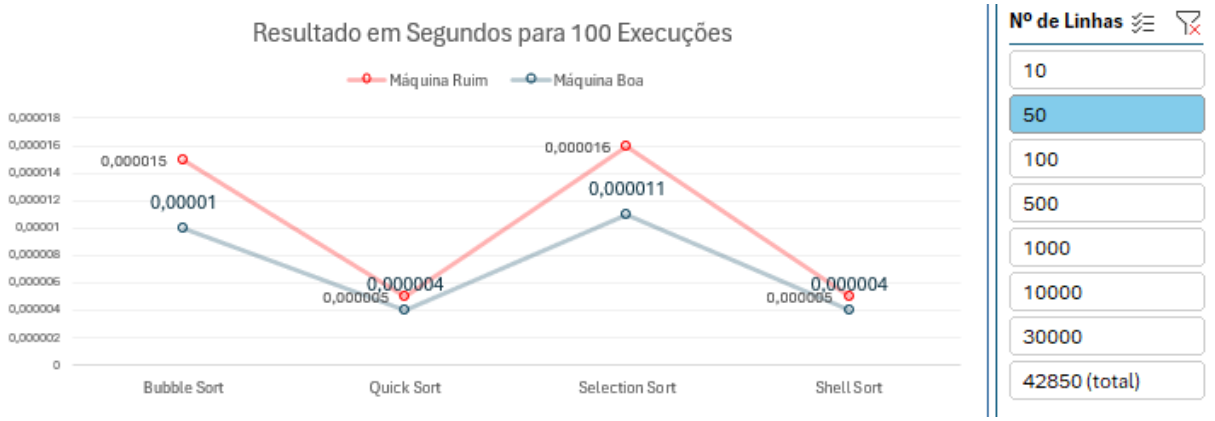




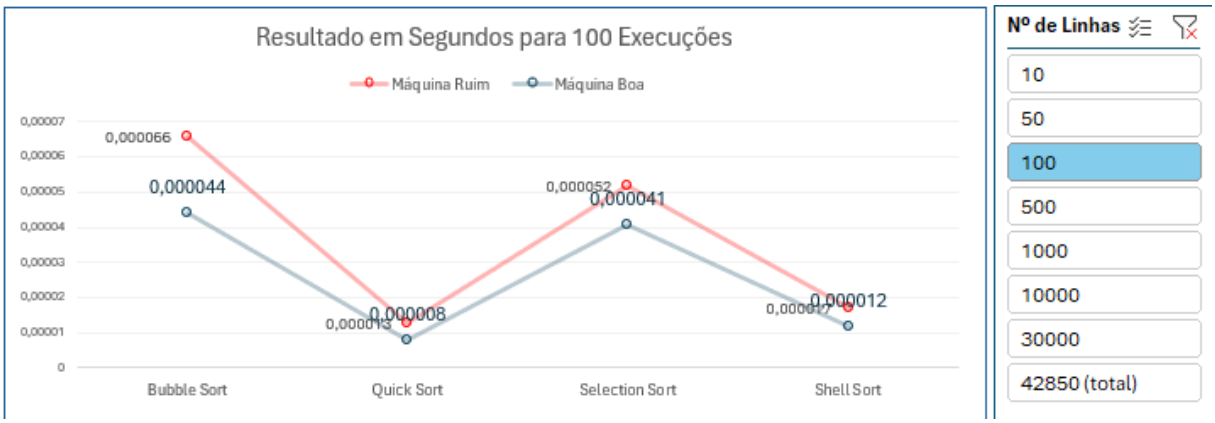
5.1.2. Tempo médio

No que quesito: tempo médio exigido para ordenação (em segundos). Podemos notar que, com 10 linhas e seguindo o padrão de 100 execuções, não há diferença **alguma** entre os desempenhos de cada algoritmo.

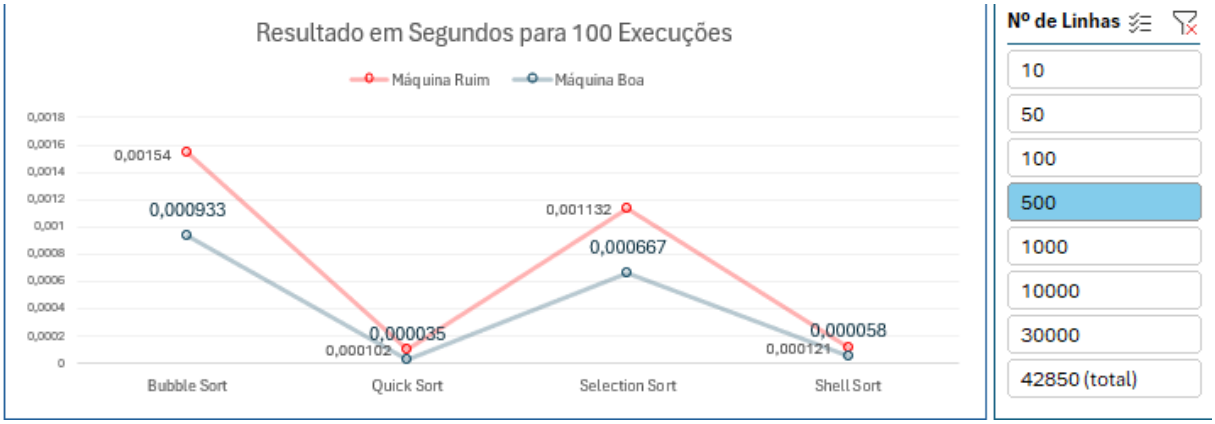
Com 50 linhas, notamos variação significativa, mas sem nenhuma novidade em termos de desempenho, se considerarmos os resultados anteriores com os tempos totais.

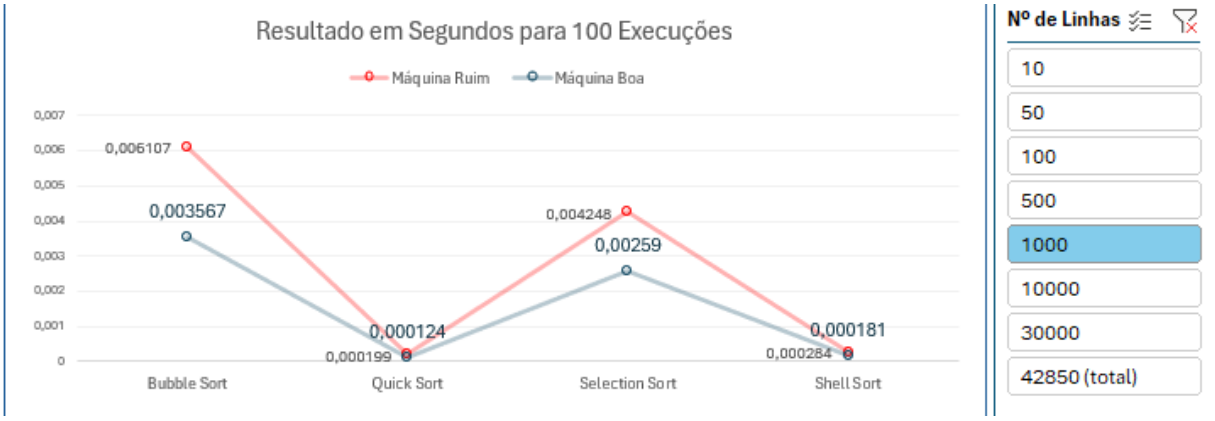


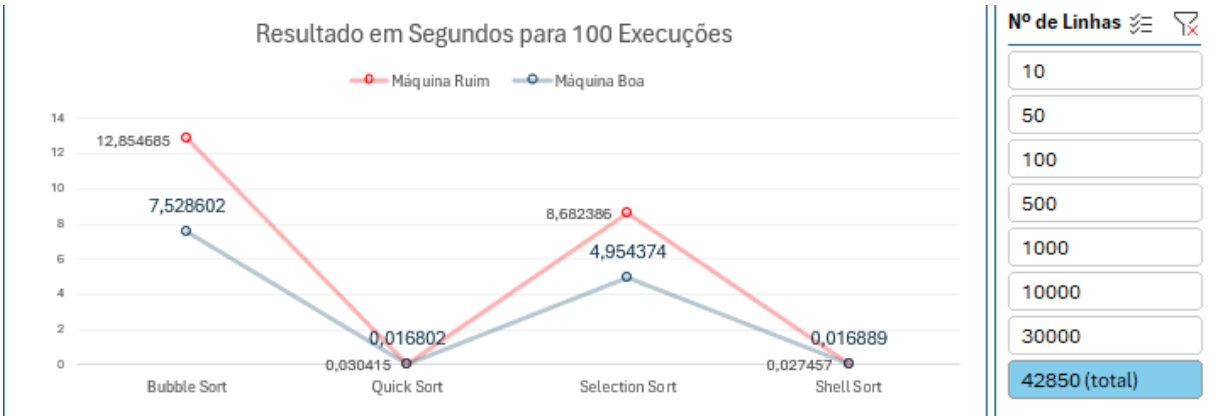
Com 100 linhas, o tempo médio dos algoritmos em ambas as máquinas, encurtam a distância.



A partir de 500 linhas, nenhuma novidade. Quick sort continua liderança em ambas as máquinas, enquanto shell demonstra ligeira desvantagem. Bubble sort se mostra o mais defasado e enquanto selection, da mesma maneira, mostra um desempenho baixo. Nenhuma novidade acima das 40000.





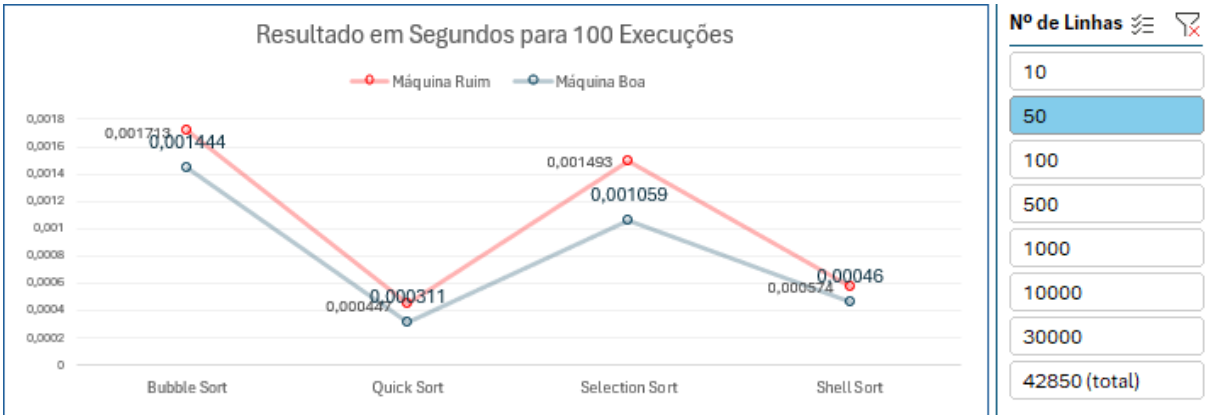


5.2. Bateria de testes com dados totalmente desordenados

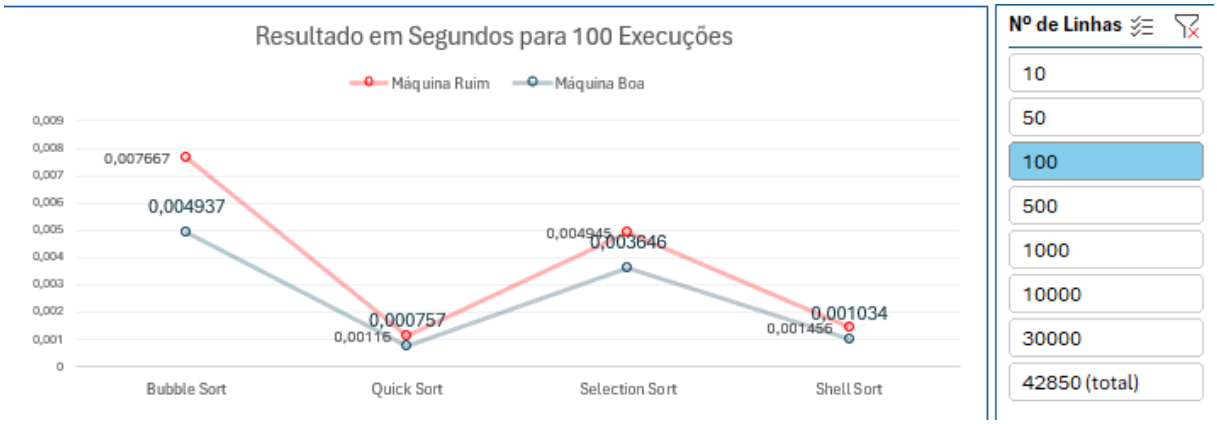
Agora, iremos expor os resultados com os dados totalmente desordenados.

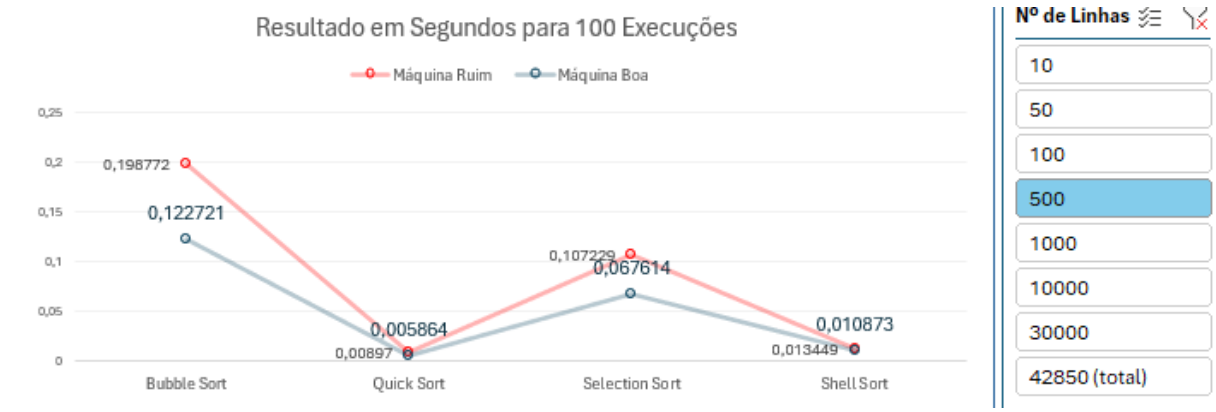
5.2.1 Tempo total

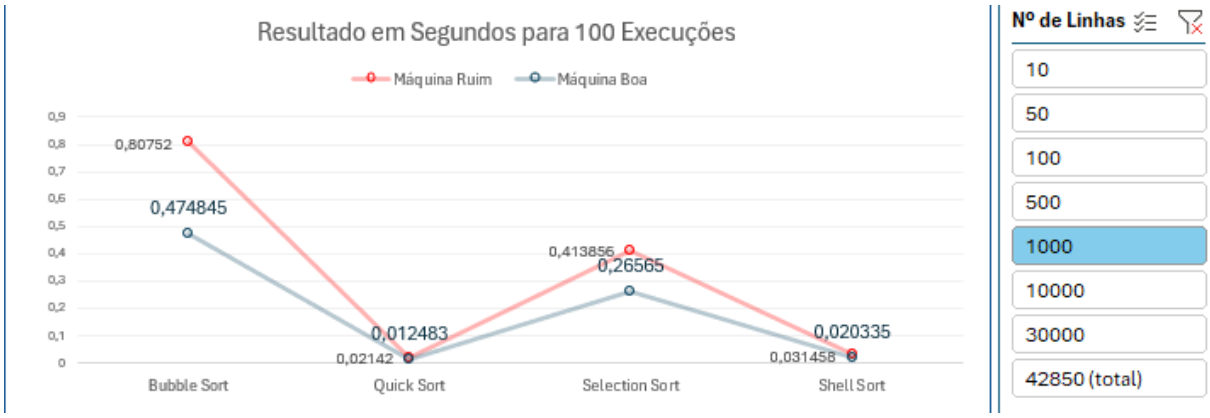
Analisando o tempo total, logo de início com 50 linhas, notamos a ligeira vantagem do quick sort em comparação com o shell sort, em ambas as máquinas. Enquanto isso, bubble sort e selection sort ficam para trás.



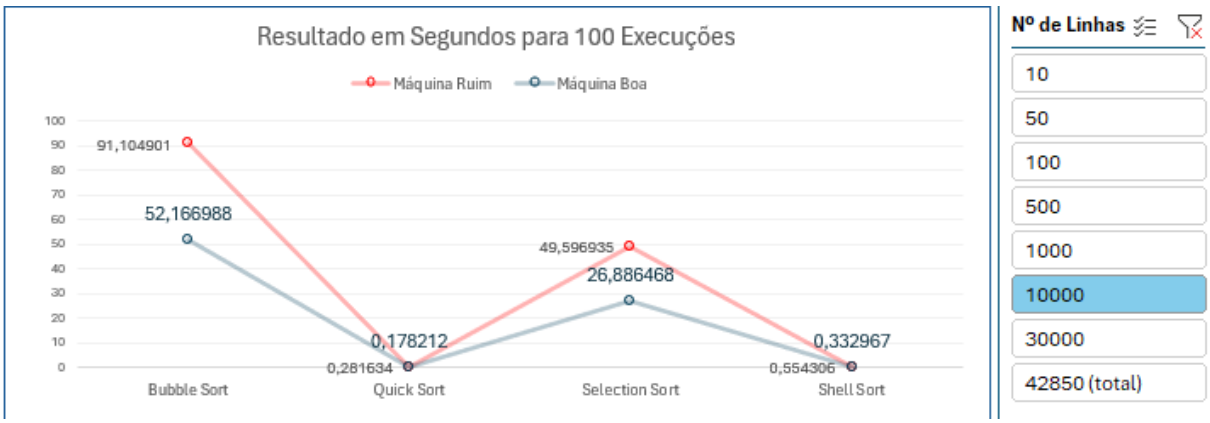
De 100 a 1000 linhas, notamos poucas diferenças.





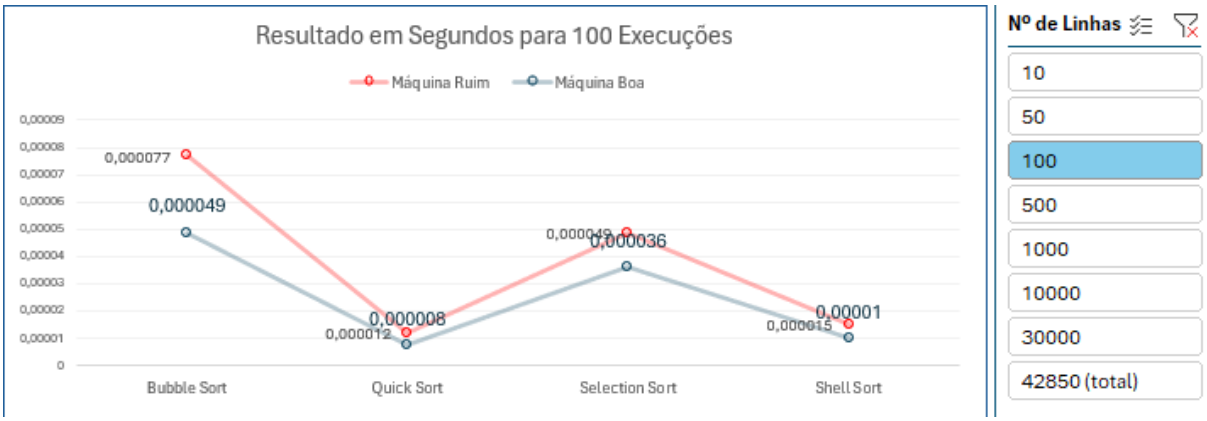


A partir de 10000, notamos maior defasagem do selection sort.

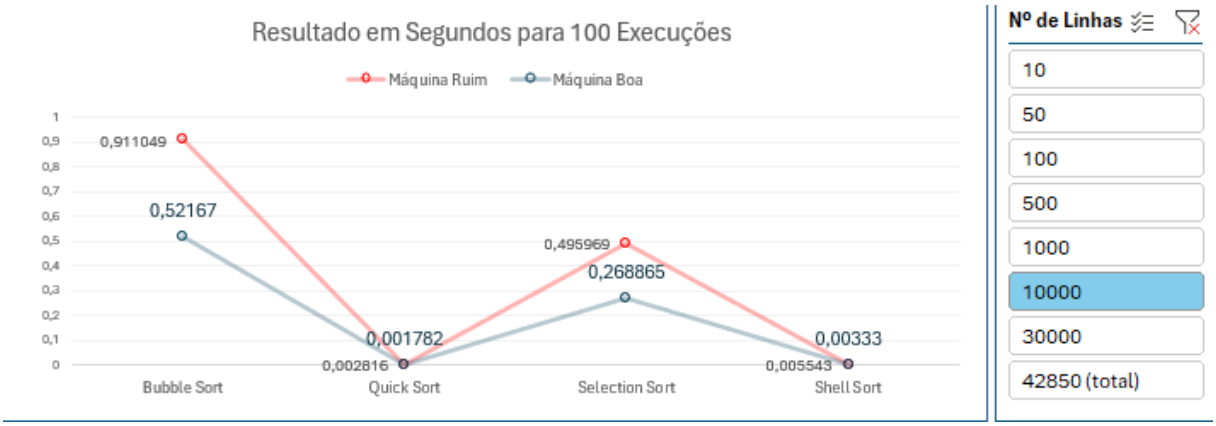


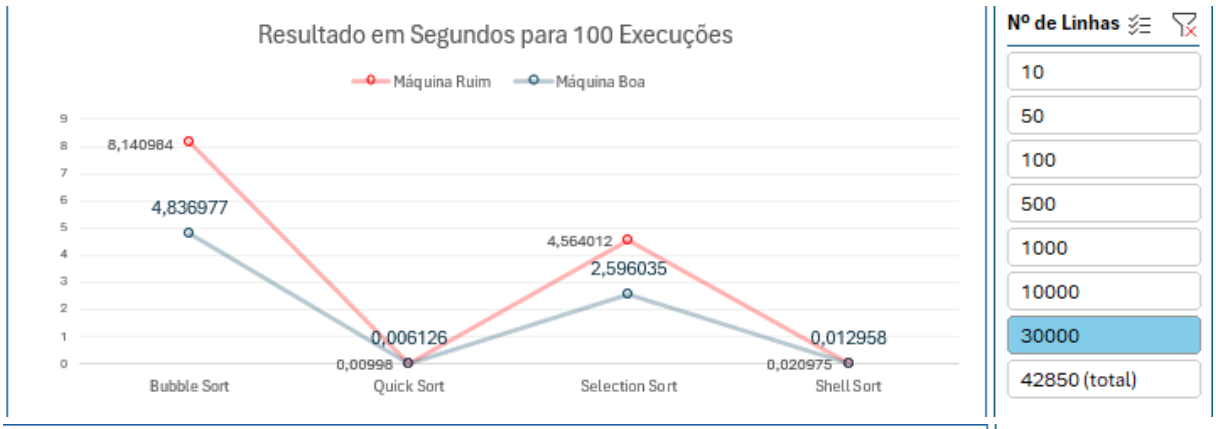
5.2.2 Tempo médio

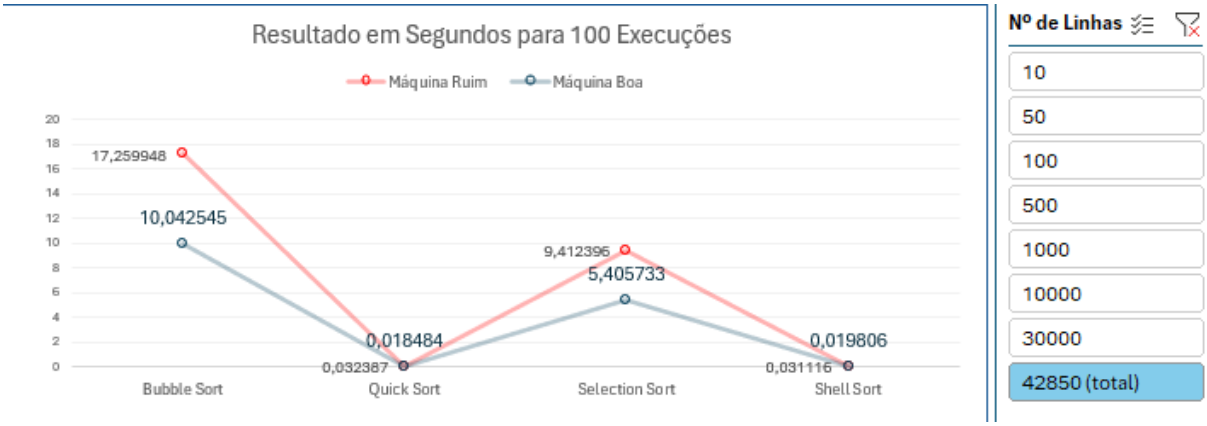
Quanto ao tempo de ordenação médio entre todas as execuções, logo com 50 linhas, observamos novamente o quick sort largando vantagem em comparação com os demais, dessa, um pouco mais à frente do shell sort. Notamos poucas diferenças com 100 linhas, mas o padrão segue o mesmo.



Nas demais induções, o padrão segue o mesmo. Notamos uma ampliação na diferença de desempenho entre os algoritmos a partir das







6. CONSIDERAÇÕES FINAIS

Ao analisar os resultados de todos os algoritmos, é importante frisarmos o contexto em que eles foram executados, e o perfil dos dados que eles estavam organizando. Iremos observar o tempo total que os algoritmos levaram para ser executados 100 vezes, nos dois estados possíveis dos dados: semi-ordenados e desordenados, sendo eles executados tanto na pior máquina virtual que possuíamos quanto na melhor, por fim concluindo uma análise quantitativa dos dados obtidos observando os principais pontos que levaram os algoritmos a obterem a performance que está sendo apresentada.

Na figura abaixo, é possível observar os resultados coletados com a base de dados no estado semi-ordenado:

Nº de Linhas	Bubble Sort	Quick Sort	Selection Sort	Shell Sort
10	0,000112	0,000148	0,000134	0,000083
50	0,001464	0,000452	0,00163	0,000545
100	0,006632	0,001339	0,005211	0,001677
500	0,153989	0,01022	0,113153	0,012061
1000	0,610657	0,019916	0,424757	0,028402
10000	70,352265	0,266333	46,48323	0,478389
30000	623,808216	1,027374	425,567169	1,818266
42850 (total)	1285,468451	3,041538	868,238606	2,745689

As barras azuis, comparam os valores no sentido horizontal, indicando visualmente qual algoritmo performou melhor em relação ao número de amostras (número de linhas). É possível facilmente perceber os algoritmos que obtiveram o pior resultado, sendo eles o Bubble Sort e o Selection Sort, onde em sua grande maioria possuem os maiores tempos de execução. O Bubble Sort por exemplo, possui um tempo de execução de 1285,5 segundos para a amostragem total (aproximadamente 20 minutos de execução), o tornando o algoritmo mais lento dentre todos os selecionados devido a sua exaustiva passagem por todos os valores repetidas vezes no processo de ordenação.

Observando o resultado com a maior amostra de dados possível, o algoritmo com o melhor tempo é o Shell Sort, obtendo um tempo de execução de 2,75 segundos, sendo interessante observar que o Quick Sort ficou com apenas 0,29 segundos de diferença, também sendo uma opção válida. Ambos esses algoritmos possuem uma execução recursiva, encurtando o “caminho” em que eles devem percorrer entre os dados através da comparação do elemento atual em relação aos outros que foram

previamente ordenados. Apesar da eficácia desta estratégia, o algoritmo acaba consumindo muito mais recursos da máquina em que está sendo executado, justificando a diferença de tempo entre os dois primeiros colocados.

Com a base de dados no estado desordenado, é possível realizar as mesmas observações:

Nº de Linhas	Bubble Sort	Quick Sort	Selection Sort	Shell Sort
10	0,000107	0,00011	0,000113	0,000121
50	0,001713	0,000447	0,001493	0,000574
100	0,007667	0,00116	0,004945	0,001456
500	0,198772	0,00897	0,107229	0,013449
1000	0,80752	0,02142	0,413856	0,031458
10000	91,104901	0,281634	49,596935	0,554306
30000	814,098415	0,998009	456,401173	2,097487
42850 (total)	1725,994806	3,238717	941,239582	3,111575

Analisando os resultados obtidos com os dados em estado semi-ordenados da máquina com boas especificações, possuímos uma leve diferença – diferença esta que é o suficiente para inverter os primeiros colocados:

Nº de Linhas	Bubble Sort	Quick Sort	Selection Sort	Shell Sort
10	0,000178	0,00016	0,000197	0,000151
50	0,001011	0,000397	0,001084	0,000413
100	0,004445	0,000813	0,004136	0,001235
500	0,093288	0,003511	0,066701	0,00578
1000	0,356651	0,012409	0,258976	0,018086
10000	39,341659	0,169772	25,783665	0,288344
30000	357,741687	0,611433	237,318285	1,057853
42850 (total)	752,860171	1,680155	495,437436	1,68887

O Quick Sort passa a ser o algoritmo mais rápido com apenas 0,008 segundos de diferença em relação ao Shell Sort, com ela crescendo quando analisamos os dados em estado desordenado – sendo 0,13 segundos a menos que o Shell Sort.

Nº de Linhas	Bubble Sort	Quick Sort	Selection Sort	Shell Sort
10	0,00012	0,000118	0,000122	0,000126
50	0,001444	0,000311	0,001059	0,00046
100	0,004937	0,000757	0,003646	0,001034
500	0,122721	0,005864	0,067614	0,010873
1000	0,474845	0,012483	0,26565	0,020335
10000	52,166988	0,178212	26,886468	0,332967
30000	483,697659	0,612591	259,603539	1,2958
42850 (total)	1004,254479	1,848362	540,573348	1,980615

Deste modo, é possível perceber o quanto as especificações das máquinas influenciam na performance do algoritmo, podendo indicar uma relação exponencial entre ela e a eficácia de um algoritmo com execução mais custosa como o Quick Sort. O estado e volume dos dados também são fatores chaves, podendo ter uma implicação direta no resultado, dependendo da forma em que o algoritmo é executado.

7. REFERENCIAS BIBLIOGRÁFICAS

JUNIOR, Elemar. **O que é e como funciona o BubbleSort (passo-a-passo)**. Video. Disponível em: https://www.youtube.com/watch?v=8RkZoBZNNgI. Acesso em: 19 ago. 2023.

PRODUCTPLAN. **BubbleSort**. Blog. Disponível em: https://www.productplan.com/glossary/bubble-sort/. Acesso em: 19 ago. 2023.

WIKIPEDIA. **Bubble Sort**. Artigo. Disponível em: https://pt.wikipedia.org/wiki/Bubble_sort. Acesso em: 19 ago. 2023.

TRYBE. **Bubble Sort: o que é e como usar? Exemplos práticos!**. Blog. Disponível em: https://blog.betrybe.com/tecnologia/bubble-sort-tudo-sobre/. Acesso em: 19 ago. 2023.

WIKIPEDIA. **Quick Sort**, Artigo. Disponível em: https://pt.wikipedia.org/wiki/Quicksort. Acesso em: 19 ago. 2023.

BRAGA, Henrique. **Algoritmos de Ordenação: Selection Sort**. Blog. Disponível em: https://henriquebraga92.medium.com/algoritmos-de-ordena%C3%A7%C3%A3o-ii-selection-sort-8ee4234deb10. Acesso em: 19 ago. 2023.

WIKIPEDIA. **Selection Sort**. Artigo. Disponível em: https://pt.wikipedia.org/wiki/Selection_sort. Acesso em: 19 ago. 2023.

SIMPLE2CODE. **Shell Sort in Java**. Blog. Disponível em: https://simple2code.com/java-programs/shell-sort-in-java/. Acesso em: 19 set. 2023.

WIKIPEDIA. **Shell Sort**. Artigo. Disponível em: https://pt.wikipedia.org/wiki/Shell_sort. Acesso em: 19 set. 2023.

SALES, Gustavo Carvalho. **Análise de desempenho de algoritmos de ordenação**. Artigo. Disponível em: https://pt.slideshare.net/gtsalles/artigo-paa-gustavo-de-carvalho-sales. Acesso em 07 out. 2023.

OLIVEIRA, Gerson Pastre de. **Pesquisa em Educação e Educação Matemática**. Editora CRV. Curitiba, 2019.

ASTRACHAN, Owen. **Bubble Sort: An Archaeological Algorithmic Analysis**. Duke University, 2019.

P@T NEWS. **História da Computação**. Blog. Disponível em: http://www.dsc.ufcg.edu.br/~pet/jornal/abril2013/materias/historia_da_computacao.html. Acesso em: 07 out. 2023.

GEEKSFORGEEKS. **Quick Sort**. Blog. Disponível em: https://www.geeksforgeeks.org/quick-sort/. Acesso em: 07 out. 2023

8. CÓDIGO FONTE

8.1 Import das Bibliotecas

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <locale.h>
```

8.2 Algoritimos de Ordenação

Nessa seção do código, fizemos as implementação dos algoritmos de ordenação escolhidos (Bubble Sort, Selection Sort, Shell Sort e Quick Sort).

8.2.1 Bubble Sort

```
// Função para ordenar um vetor de strings usando o Bubble Sort
void bubbleSort(char **arr, int n) {
            int i,j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (strcmp(arr[j], arr[j + 1]) > 0) {
                // Troca as strings se estiverem na ordem errada
                char *temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

8.2.2 Selection Sort

```
// Função para ordenar um vetor de strings usando o Selection Sort
void selectionSort(char **arr, int n) {
    int i, j, min_index;
    char *temp;

    for (i = 0; i < n - 1; i++) {
        min_index = i;
        for (j = i + 1; j < n; j++) {
            if (strcmp(arr[j], arr[min_index]) < 0) {
                min_index = j;
            }
        }

        // Troca as strings se estiverem na ordem errada
        if (min_index != i) {
            temp = arr[i];
            arr[i] = arr[min_index];
            arr[min_index] = temp;
        }
    }
}
```

8.2.3 Shell Sort

```
// Função para ordenar um vetor de strings usando o Shell Sort
void shellSort(char **arr, int n) {
    int gap, i, j;
    char *temp;

    for (gap = n / 2; gap > 0; gap /= 2) {
        for (i = gap; i < n; i++) {
            temp = arr[i];
            for (j = i; j >= gap && (strcmp(arr[j - gap], temp) > 0);j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

8.2.4 Quick Sort

```
// Função de partição usada no Quick Sort
int partition(char **arr, int low, int high) {
    char *pivot = arr[high];
    int i = low - 1,j;

    for (j = low; j <= high - 1; j++) {
        if (strcmp(arr[j], pivot) < 0) {
            i++;
            char *temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    char *temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

// Função principal do Quick Sort
void quickSort(char **arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

8.3 Função duplicaArray

Utilizada para fazer a cópia do array img[] , permitindo executar o algoritmo de ordenação n vezes (n definido pelo usuário). Já que o algoritmo de ordenação irá ordenar o array cópia deixando o array original sem modificações.

```
//Duplica o Array Original
char** duplicaArray(char** original, int size)
{
  char** copia = (char**)malloc(size * sizeof(char*));
  int i;
  for (i = 0; i < size; i++) {
    copia[i] = (char*)malloc(strlen(original[i]) + 1);
    strcpy(copia[i], original[i]);
  }
  return copia;
}
```

8.4 Função imprimeArquivo

Essa função tem como objetivo imprimir o array ordenado (com os nomes dos arquivo de imagens), em um arquivo txt de nome imgs_ord.txt.

```
//Função Imprime Arquivo
void imprimeArquivo(char **imgCopia,int img_size)
{
  int opt2,i;
  printf("\nDeseja imprimir a lista de imagens ordenadas em um arquivo de texto?\n1 - Sim\n2 -
Não\n");
  scanf("%d", &opt2);

  if (opt2 == 1) {
    FILE *arquivo_saida = fopen("imgs_ord.txt", "w");
    if (arquivo_saida == NULL) {
      perror("Erro ao criar o arquivo de saída");
      exit(EXIT_FAILURE);
    }

    for (i = 0; i < img_size; i++) {
      fprintf(arquivo_saida, "%s\n", imgCopia[i]);
    }

    fclose(arquivo_saida);
    printf("As imagens ordenadas foram salvas em 'imgs_ord.txt'.\n");

    // Libera a memória alocada para imgCopia
    for (i = 0; i < img_size; i++) {
      free(imgCopia[i]);
    }
    free(imgCopia);
  }
}
```

8.5 Função main

Aqui temos nossa função principal, na qual ocorrem os seguintes passos:

Leitura do arquivo de texto com os nomes dos arquivos das imagens desordenados. Colocando o valor de cada linha em uma posição de um array de strings.

Seleção do algoritmo a ser utilizado para fazer a ordenação dos dados: Por meio de um switch case com a variável opt a qual o valor é definido pelo usuário.

Seleção de quantas vezes o algoritmo de ordenação deverá ser executado.

Depois disso, a função do algoritmo de ordenação correspondente é chamada entre duas funções clock(), para medir o tempo de ordenação dos dados.

Ao finalizar a ordenação, é mostrada ao usuário a opção de salvar os dados ordenados em um arquivo. Caso o usuário opte por realizar o salvamento, então é chamada a função imprimeArquivo que salva o conteúdo do array ordenado no arquivo de texto imgs_ord.txt.

Por fim é realizada a desalocação do array imgCopia.

```
void main() {
  setlocale(LC_ALL,"Portuguese");
  FILE *fp;
  char line[BUFSIZ];  // Buffer para armazenar uma linha
  char **img = NULL;  // Vetor para armazenar as linhas
  int img_size = 0,i;

  fp = fopen("wildfire.txt", "r");
  if (fp == NULL) {
    perror("Erro ao abrir o arquivo!");
    exit(EXIT_FAILURE);
  }

  while (fgets(line, sizeof(line), fp) != NULL) {
    // Remove o caractere de nova linha do final da linha, se presente
    char *newline = strchr(line, '\n');
    if (newline != NULL) {
      *newline = '\0';
    }

    // Aloca memória para uma nova string no vetor img
    img = (char **)realloc(img, (img_size + 1) * sizeof(char *));
    img[img_size] = (char *)malloc(strlen(line) + 1);

    // Copia a linha atual para o vetor img
    strcpy(img[img_size], line);
    img_size++;
  }

  int opt, opt2, numRuns,run;
  clock_t inicio, fim;
  double tempo, soma_t;
  do{
    printf("\n#### Algoritmos de Ordenação Disponíveis ####\n");
    printf("1 - Bubble Sort\n");
    printf("2 - Selection Sort\n");
    printf("3 - Shell Sort\n");
    printf("4 - Quick Sort\n");
    printf("Escolha a opção desejada: ");
    scanf("%d", &opt);

    switch (opt) {
      case 1:
        //Bubble Sort
        printf("Quantas vezes deseja rodar o algoritmo? ");
```

```
    scanf("%d", &numRuns);
    printf("\nExecutando...");
    if(numRuns > 10){
      printf("\nNúmero de execuções excede 10, iremos mostrar somente os resultados");
    }

    for (run = 0; run < numRuns; run++) {
      char** imgCopia = duplicaArray(img, img_size);

      inicio = clock();
      bubbleSort(imgCopia, img_size);
      fim = clock();

      tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;
      soma_t += tempo;
      if(numRuns <= 10){
        printf("\nTempo para ordenar com Bubble Sort (Execução %d): %f segundos\n", run + 1, tempo);
      }

      if(run + 1 == numRuns){
        printf("\nTempo total: %f",soma_t);
        printf("\nTempo médio: %f",soma_t/numRuns);
        imprimeArquivo(imgCopia, img_size);
      }
      else{
      // Libera a memória alocada para imgCopia
        for (i = 0; i < img_size; i++) {
          free(imgCopia[i]);
        }
        free(imgCopia);
      }
    }
  break;

  case 2:
    //Selection Sort
    printf("Quantas vezes deseja rodar o algoritmo? ");
    scanf("%d", &numRuns);
    printf("\nExecutando...");
    if(numRuns > 10){
      printf("\nNúmero de execuções excede 10, iremos mostrar somente os resultados");
    }

    for (run = 0; run < numRuns; run++) {
      char** imgCopia = duplicaArray(img, img_size);

      inicio = clock();
      selectionSort(imgCopia, img_size);
      fim = clock();

      tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;
      soma_t += tempo;
      if(numRuns <= 10){
        printf("\nTempo para ordenar com Selection Sort (Execução %d): %f segundos\n", run + 1, tempo);
      }

      if(run + 1 == numRuns){
        printf("\nTempo total: %f",soma_t);
        printf("\nTempo médio: %f",soma_t/numRuns);
        imprimeArquivo(imgCopia, img_size);
      }
      else{
        // Libera a memória alocada para imgCopia
        for (i = 0; i < img_size; i++) {
          free(imgCopia[i]);
        }
        free(imgCopia);
```

```
    }
  }
break;

case 3:
  // Shell Sort
  printf("Quantas vezes deseja rodar o algoritmo? ");
  scanf("%d", &numRuns);
  printf("\nExecutando...");
  if(numRuns > 10){
    printf("\nNúmero de execuções excede 10, iremos mostrar somente os resultados");
  }

  for (run = 0; run < numRuns; run++) {
    char** imgCopia = duplicaArray(img, img_size);

    inicio = clock();
    shellSort(imgCopia, img_size);
    fim = clock();

    tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;
    soma_t += tempo;
    if(numRuns <= 10){
      printf("\nTempo para ordenar com Shell Sort (Execução %d): %f segundos\n", run + 1, tempo);
    }

    if(run + 1 == numRuns){
      printf("\nTempo total: %f",soma_t);
      printf("\nTempo médio: %f",soma_t/numRuns);
      imprimeArquivo(imgCopia, img_size);
    }
    else{
      // Libera a memória alocada para imgCopia
      for (i = 0; i < img_size; i++) {
          free(imgCopia[i]);
      }
      free(imgCopia);
    }
  }
break;

case 4:
  // Quick Sort
  printf("Quantas vezes deseja rodar o algoritmo? ");
  scanf("%d", &numRuns);
  printf("\nExecutando...");
  if(numRuns > 10){
    printf("\nNúmero de execuções excede 10, iremos mostrar somente os resultados");
  }

  for (run = 0; run < numRuns; run++) {
    char** imgCopia = duplicaArray(img, img_size);

    inicio = clock();
    quickSort(imgCopia, 0, img_size - 1);
    fim = clock();

    tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;
    soma_t += tempo;
    if(numRuns <= 10){
      printf("\nTempo para ordenar com Quick Sort (Execução %d): %f segundos\n", run + 1, tempo);
    }

    if(run + 1 == numRuns){
      printf("\nTempo total: %f",soma_t);
      printf("\nTempo médio: %f",soma_t/numRuns);
      imprimeArquivo(imgCopia, img_size);
```

```
        }
        else{
          // Libera a memória alocada para imgCopia
          for (i = 0; i < img_size; i++) {
            free(imgCopia[i]);
          }
          free(imgCopia);
        }
      }
      break;

      default:
        printf("Escolha inválida.\n");
    }
  } while(opt > 4 || opt < 0);


  // Libera a memória alocada para as strings no vetor img
  for (i = 0; i < img_size; i++) {
    free(img[i]);
  }

  // Libera o vetor img
  free(img);
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```

9. FICHA APS

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos Individuais e em grupo, práticas de ensino e outras)

NOME: Leonardo de Souza Rodrigues

RA: F344HB-2 CURSO: Ciência da Computação

CAMPUS: Jundiaí SEMESTRE: 4º Semestre TURNO: Noturno

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA	
			ALUNO	PROFESSOR
09/08	Reunião para definição do tema	2	Leonardo S. Rodrigues	
19/08	Pesquisa de referências	8	Leonardo S. Rodrigues	
26/08	Divisão das Tarefas e responsabilidades	4	Leonardo S. Rodrigues	
02/09	Reunião para iniciar as implementação	8	Leonardo S. Rodrigues	
16/09	Pesquisa dos algoritmos mais utilizados	8	Leonardo S. Rodrigues	
30/09	Implementação dos algoritmo	10	Leonardo S. Rodrigues	
07/10	Pesquisa de referência	6	Leonardo S. Rodrigues	
14/10	Início da documentação	10	Leonardo S. Rodrigues	
15/10	Sinalização do código	10	Leonardo S. Rodrigues	
21/10	Realização dos testes	6	Leonardo S. Rodrigues	
28/10	Finalização da Documentação	5	Leonardo S. Rodrigues	
04/11	Finalização das referências	3	Leonardo S. Rodrigues	

TOTAL DE HORAS: 80 Horas

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos Individuais e em grupo, práticas de ensino e outras)

NOME: Olga Faria de Almeida

RA: N805902 CURSO: Ciência da Computação

CAMPUS: Jundiaí SEMESTRE: 4º TURNO: Noturno

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA	
			ALUNO	PROFESSOR
09/08	Reunião para definição do tema	2		
19/08	Pesquisa de Referências I	8		
26/08	Divisão das tarefas e responsabilidades	4		
02/09	Reunião para iniciar as implementações	8		
16/09	Pesquisa dos algoritmos mais utilizados	8		
30/09	Implementação dos algoritmos	10		
07/10	Pesquisa de Referências II	6		
14/10	Início da Documentação	10		
15/10	Sinalização dos Códigos	10		
21/10	Realização dos testes	6		
28/10	Finalização da Documentação	5		
04/11	Finalização das Referências	3		

TOTAL DE HORAS: 80

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)

NOME: Vitor dos Santos Rosa

RA: G521GE3 CURSO: Ciência da Computação

CAMPUS: Jundiaí SEMESTRE: 4º Semestre TURNO: Noite

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA ALUNO	ASSINATURA PROFESSOR
09/08	Reunião para definição do Tema	2		
19/08	Pesquisa de Referências I	8		
26/08	Divisão das Tarefas e Responsabilidades	4		
02/09	Reunião Para criar os implementações	2		
16/09	Pesquisa dos algoritmos mais utilizados	8		
30/09	Implementação dos algoritmos	10		
07/10	Pesquisa de referências II	6		
14/10	Início da documentação	10		
15/10	Finalização dos códigos	10		
21/10	Realização dos testes	6		
28/10	Finalização da documentação	5		
04/11	Finalização das referências	3		

TOTAL DE HORAS: 80

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratorios, atividades em biblioteca, Iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)

NOME: KAIKY DE LARA SALES

RA: N9718H8 CURSO: CIÊNCIA DA COMPUTAÇÃO

CAMPUS: JUNDIAI SEMESTRE: 4º SEMESTRE TURNO: NOITE

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA ALUNO	ASSINATURA PROFESSOR
09/08	REUNIÃO PARA DEFINIÇÃO DOS TEMAS	2		
19/08	PESQUISA DE REFERÊNCIA I	8		
26/08	DIVISÃO DAS TAREFAS E RESPONSABILIDADES	4		
02/09	REUNIÃO PARA CRIAR AS IMPLEMENTAÇÕES	8		
16/09	PESQUISA DOS ALGORITIMOS MAIS UTILIZADOS	8		
30/09	IMPLEMENTAÇÃO DOS ALGORITIMOS	10		
07/10	PESQUISA DE REFERENCIA II	6		
14/10	INICIO DA DOCUMENTAÇÃO	10		
15/10	FINALIZAÇÃO DOS CODIGOS	10		
21/10	REALIZAÇÃO DOS TESTES	6		
28/10	FINALIZAÇÃO DA DOCUMENTAÇÃO	5		
04/11	FINALIZAÇÃO DAS REFERENCIAS	3		

TOTAL DE HORAS: 80 HORAS

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)

NOME: Nídia Pimenta da Silva

RA: N363579 CURSO: Ciência da computação

CAMPUS: Fortaleza SEMESTRE: 4º semestre TURNO: DIURNO

DATA	ATIVIDADE	TOTAL DE HORAS	ASSINATURA	
			ALUNO	PROFESSOR
09/09	Reunião para definição de Temas	2	Nídia Pimenta	
19/09	Pesquisa de referência I	8	Nídia Pimenta	
26/09	Divisão das tarefas e Responsabilidades	4	Nídia Pimenta	
03/09	Reunião para criar as Implementações	8	Nídia Pimenta	
16/09	Pesquisa dos algoritmos mais utilizados	9	Nídia Pimenta	
30/09	Implementação dos algoritmos	10	Nídia Pimenta	
07/10	Pesquisa de Referência II	6	Nídia Pimenta	
10/10	Revisão da documentação	10	Nídia Pimenta	
15/10	Finalização dos códigos	10	Nídia Pimenta	
21/10	Realização dos testes	6	Nídia Pimenta	
29/10	Finalização da Documentação	5	Nídia Pimenta	
04/11	Finalização das Referências	3	Nídia Pimenta	

TOTAL DE HORAS: 80