Alex Magalhães da Silva Junior
Allan Custódio Diniz Marques

Trabalho Prático
Algoritmos de Ordenação e Busca

Relatório técnico de trabalho prático solicitado pelo professor Rodrigo Hübner na disciplina de Algoritmos e Estrutura de Dados 1 do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Junho / 2025

Resumo

Este relatório apresenta a implementação, comparação e análise de desempenho dos algoritmos de ordenação SelectionSort (padão e otimizado), BubbleSort (padrão e otimizado), InsertionSort, bem como dos algoritmos de busca sequencial e binária. O estudo foi conduzido utilizando arquivos binários contendo conjuntos de dados do tipo flutuante de diferentes tamanhos (pequeno, médio e grande), calibrados para refletir tempos de execução médios de 1 segundo, 30 segundos e 3 minutos, respectivamente. Os resultados incluem medições de tempo de execução, número de comparações e trocas, além de gráficos comparativos gerados para visualização dos dados. A análise crítica destaca as diferenças de desempenho entre os algoritmos, as vantagens das versões otimizadas e a eficiência da busca binária em relação à sequencial. Cumprindo os objetivos propostos, na seção 4 realizaremos as análises completas dos algoritmos de ordenação juntamente com os resultados das buscas binárias e sequenciais.

Palavras-chave: Algoritmos. Ordenação. Buscas. Sequencial. Binária.

Sumário

1 Introdução

Algoritmos de ordenação e busca são fundamentais na ciência da computação, impactando diretamente a eficiência de sistemas e aplicações. Este trabalho prático visa explorar o comportamento de algoritmos clássicos de ordenação (SelectionSort, BubbleSort e InsertionSort, em suas versões padrão e otimizadas) e de busca (sequencial e binária), utilizando dados armazenados em arquivos binários. A escolha dos tamanhos dos conjuntos de dados foi baseada em testes preliminares, garantindo que as execuções reflitam cenários reais de uso.

Além da implementação, o relatório aborda a análise comparativa dos algoritmos, considerando métricas como tempo de execução e número de operações, e apresenta os resultados por meio de visualizações gráficas. A discussão crítica destaca os trade-offs entre os métodos, como a relação entre complexidade teórica e desempenho prático, e as condições em que as otimizações se mostram vantajosas.

2 Objetivos

- Implementar e analisar algoritmos de ordenação.
- Avaliar métodos de busca em diferentes cenários.
- Comparar o desempenho dos algoritmos por meio de métricas empíricas.
- Gerar visualizações gráficas para ilustrar as eficiências relativas.
- Discutir sobre desvantagens e vantagens dos algoritmos e cenários ideiais para seus usos.

3 Materiais

- Visual Studio Code 1.100.2
- Documento PDF da avaliação disponibilizado pelo professor.
- Máquina utilizada para os testes:
    1. Acer Aspire 5
    2. Intel® Core™ i5-10210U CPU @ 1.60 GHz × 4x;
    3. 8GB RAM 2666 Mhz DDR4;
    4. 512GB SSD sata;
    5. Sistema: Ubuntu 24.04.02 LTS x86-64;
    6. Kernel Linux 6.11.10-26-generic;

4 Procedimentos e Resultados

4.1 Gerador de Arquivos Binários

Um gerador de dados foi implementado na linguagem C++, juntamente com grande parte das implementações deste projeto, para povoar arquivo binários de dados separados em pequeno, médio e grande. O objetivo desta implementação foi seguir a risca o enunciado proposto pela atividade prática para garantir que os métodos de ordenação em sua totalidade de tempo de respsta retornem em tempos de 1 segundo, 30 segundos e 3 minutos para os arquivos pequeno, médio e grande respectivamente.

A implementação do gerador conta com um laço iterativo que gera números flutuantes entre -1 milhão e +1 milhão, armazenando seus resultados em cada arquivo binário nomeado com a função arquivo.write("Pasta de destino", valor flutuante).

Seguindo as especificações mencionadas na atividade prática, para tanto, foram gerados para o arquivo pequeno.bin um total de 25.000 números flutuantes, garantindo 1 segundo de execução dos algoritmos de ordenação e, junto da mesma implementação, garantindo o arquivo medio.bin com 110.000 números flutuantes com 30 segundos de execução e grande.bin com 270.000 floats e 190 segundos de execução (pouco mais de 3 minutos).

```cpp
// Função que cria um arquivo binário com 'tamanho' números float aleatórios.
void criar_arquivo(const std::string& caminho, size_t tamanho) {
    // Abre o arquivo para escrita em modo binário.
    std::ofstream arquivo(caminho, std::ios::binary);

    // Verifica se o arquivo foi aberto corretamente.
    if (!arquivo.is_open()) {
        std::cerr << "Erro ao criar o arquivo '" << caminho << "': "
                  << std::strerror(errno) << std::endl;
        return; // Encerra a função se não conseguiu abrir o arquivo.
    }

    // Inicializa o gerador de números aleatórios.
    std::random_device rd;                                  // Fonte de entropia (aleatoriedade).
    std::mt19937 gen(rd());                                 // Gerador Mersenne Twister com semente aleatória.
    std::uniform_real_distribution<float> dist(-1e6f, 1e6f); // Gera floats entre -1.000.000 e 1.000.000

    // Gera 'tamanho' números aleatórios e grava no arquivo.
    for (size_t i = 0; i < tamanho; ++i) {
        float valor = dist(gen);                                                    // Gera um número aleatório.
        arquivo.write(reinterpret_cast<const char*>(&valor), sizeof(float)); // Escreve o número no arquivo como bytes binários.
    }

    // Fecha o arquivo.
    arquivo.close();
    std::cout << "Arquivo '" << caminho << "' criado com " << tamanho << " floats." << std::endl;
}

int gerar() {
    criar_arquivo("../dados/pequeno.bin", 25'000);
    criar_arquivo("../dados/medio.bin",   110'000);
    criar_arquivo("../dados/grande.bin", 270'000);
    return 0;
}
```

Figura 1 – Código do gerador de arquivos binários

4.2 Métodos de Ordenação

Para os algoritmos de ordenação foram implementadas algumas funções auxiliares (max e troca) para mediar as execuções durante as odens dos métodos.

```
// Retorna o índice do maior elemento no intervalo [ini, fim] do vetor v.
int max(float* v, int ini, int fim, Contador& c){
    int maior = ini;
    for(int i = ini+1; i <= fim; i++){
        c.comparacoes++; // Incrementa o contador de comparações a cada comparação realizada.
        if(v[i] > v[maior])
            maior = i;
    }
    return maior;
}

// Troca os elementos nas posições a e b do vetor v.
void troca(float* v, int a, int b, Contador& c){
    float aux = v[a];
    v[a] = v[b];
    v[b] = aux;
    c.trocas++; // Incrementa o contador de trocas.
}
```

Figura 2 – Código das funções auxiliares

Após os auxiliares, temos as implementações clássicas dos algoritmos de ordem e suas versões otimizadas.

O algoritmo simples de Selection Sort, nesta implementação, seleciona repetidamente o menor elemento de uma parte não ordenada do vetor e colocando-o na posição correta. A cada iteração, ele percorre a parte não ordenada do vetor para encontrar o menor valor e, em seguida, realiza uma troca com o primeiro elemento dessa parte.

```
// Implementação do Selection Sort (Ordenação por Seleção).
// Encontra o maior elemento da parte não ordenada e o coloca na posição correta no final.
void SelectionSort(float* v, int n, Contador& c){
    for(int i = n-1; i > 0; i--){
        int pmaior = max(v, 0, i, c); // Encontra o maior elemento entre 0 e i.
        troca(v, pmaior, i, c);       // Coloca o maior na posição correta (final da parte não ordenada).
    }
}
```

Figura 3 – Código do algoritmo Selection Sort

O algoritmo otimizado de Selection Sort, nesta implementação, tem como objetivo primordial apenas considerar trocas caso os índices sejam diferentes. Esse tipo de implementação acaba sendo responsável por diminuir a quantidade de trocas realizadas durante a execução da função.

```
// Versão otimizada do Selection Sort.
// Só realiza a troca se o maior elemento não estiver já na posição correta.
void SelectionSortOpt(float* v, int n, Contador& c){
    for(int i = n-1; i > 0; i--){
        int pmaior = max(v, 0, i, c);
        if (pmaior != i) // Evita troca desnecessária
            troca(v, pmaior, i, c);
    }
}
```

Figura 4 – Código do algoritmo Selection Sort otimizado

O algoritmo simples de Bubble Sort, nesta implementaçãopares de elementos adjacentes e trocando-os se estiverem fora de ordem. Esse processo faz com que os maiores valores "subam"para o final do vetor, como bolhas em um líquido, daí o nome "bubble". A cada varredura completa, o maior valor restante se posiciona corretamente no final, e o processo se repete para o restante do vetor.

```
// Implementação do Bubble Sort (Ordenação por Flutuação).
// Compara pares de elementos adjacentes e os troca se estiverem fora de ordem.
void BubbleSort(float* v, int n, Contador& c) {
    for(int j = n-1; j > 0; j--) {
        for(int i = 0; i < j; i++) {
            c.comparacoes++; // Incrementa o contador de comparações a cada comparação realizada.
            if(v[i] > v[i + 1])
                troca(v, i, i+1, c);
        }
    }
}
```

Figura 5 – Código do algoritmo Bubble Sort

O algoritmo otimizado de Bubble Sort, nesta implementação, tem como objetivo principal encerrar a ordenação mais cedo caso o vetor já esteja ordenado. Para isso, utiliza uma variável booleana que verifica se houve alguma troca durante uma passada completa. Se nenhuma troca for realizada, isso indica que todos os elementos estão em ordem, permitindo que o algoritmo pare antes de completar todas as iterações. Essa abordagem reduz o número de comparações e trocas em vetores parcialmente ou totalmente ordenados, tornando o algoritmo mais eficiente nesses casos.

```cpp
// Versão otimizada do Bubble Sort.
// Se em uma passada completa não houver trocas, o vetor já está ordenado e o algoritmo encerra.
void BubbleSortOpt(float* v, int n, Contador& c) {
    bool trocou;
    for(int j = n - 1; j > 0; j--) {
        trocou = false;
        for(int i = 0; i < j; i++) {
            c.comparacoes++;            // Incrementa o contador de comparações a cada comparação realizada.
            if(v[i] > v[i + 1]) {
                troca(v, i, i + 1, c);
                trocou = true;
            }
        }
        if(!trocou) break;              // termina cedo se não houve trocas.
    }
}
```

Figura 6 – Código do algoritmo Bubble Sort otimizado

O algoritmo Insertion Sort, nesta implementação, tem como objetivo principal inserir cada elemento do vetor na posição correta entre os elementos já ordenados à esquerda. A cada iteração, o valor atual é comparado com os anteriores e deslocado até encontrar sua posição ideal. Essa abordagem é eficiente para vetores pequenos ou quase ordenados, pois, nesse cenário, o número de comparações e movimentações é reduzido. O algoritmo se destaca por sua simplicidade e bom desempenho em casos favoráveis, mesmo que no pior caso tenha complexidade quadrática.

É utilizado uma função auxiliar Insertion, que é responsável por inserir um único elemento do vetor (na posição k) em sua posição correta entre os elementos anteriores já ordenados. Ela percorre o vetor da posição k - 1 até o início, comparando o valor atual com os anteriores, deslocando-os para a direita sempre que forem maiores. Ao final, o elemento é inserido na posição correta. Essa função realiza o trabalho fundamental do algoritmo InsertionSort, e sua repetição para cada elemento da sequência completa a ordenação do vetor. Além disso, ela contabiliza comparações e trocas por meio do objeto Contador, permitindo análise de desempenho detalhada.

```cpp
// Função auxiliar do Insertion Sort.
// Insere o elemento v[k] na posição correta na parte ordenada do vetor (v[0..k-1]).
void insertion(float* v, int k, Contador& c) {
    float x = v[k];
    int i = k - 1;
    while(i >= 0) {
        c.comparacoes++;    // Incrementa o contador de comparações a cada comparação realizada.
        if(v[i] > x) {
            v[i+1] = v[i];  // Desloca elemento maior.
            c.trocas++;     // Incrementa o contador de trocas.
            i--;
        } else break;
    }
    v[i+1] = x;             // Insere o elemento na posição correta.
}
```

Figura 7 – Código do algoritmo auxiliar Insertion

```cpp
// Implementação do Insertion Sort (Ordenação por Inserção).
// Percorre o vetor a partir do segundo elemento, inserindo cada um na posição correta da parte já ordenada.
void InsertionSort(float* v, int n, Contador& c) {
    for (int j = 1; j < n; j++) {
        insertion(v, j, c);
    }
}
```

Figura 8 – Código do algoritmo Insertion Sort

Dentro da implementação dos algoritmos de ordenação também temos a função medirTempo, feita em C++, responsável por colher os parâmetros necessários da função de ordenação que selecionarmos na função main e, durante este processo, calcular o tempo total que o método selecionado demorar para executar.

Esta função acaba por também executar um salvamento dos dados ordenados na pasta resultados.csv que acompanha toda a implementação, nesta pasta de destino serão dispostos os resultados obtidos para então gerarmos os gráficos comparativos deste estudo.

```cpp
void medirTempo(const std::string& nomeAlg,
                void (*alg)(float*, int, Contador&),
                const std::vector<float>& dados,
                const std::string& nomeArquivo)
{
    // Cria cópia dos dados para ordenação
    std::vector<float> copia = dados;
    Contador cont;

    // Executa e mede o tempo
    auto inicio = std::chrono::high_resolution_clock::now();
    alg(copia.data(), static_cast<int>(copia.size()), cont);
    auto fim = std::chrono::high_resolution_clock::now();

    // Mostra resultados no console
    auto duracao = std::chrono::duration_cast<std::chrono::milliseconds>(fim - inicio);
    long long tempo = duracao.count();

    std::cout << "\n┌──────────────────────────────────────┐" << std::endl;
    std::cout << "   Resultados do " << nomeAlg << std::endl;
    std::cout << "├──────────────────────────────────────┤" << std::endl;
    std::cout << "   Tempo de execução: " << tempo << "ms" << std::endl;
    std::cout << "   Comparações:      " << cont.comparacoes << std::endl;
    std::cout << "   Trocas:           " << cont.trocas << std::endl;
    std::cout << "└──────────────────────────────────────┘" << std::endl;

    // Salva os resultados no arquivo resultado.csv para geração dos gráficos.
    std::ofstream saida("resultados.csv", std::ios::app);
    if (saida.is_open()) {
        saida << nomeAlg << "," << nomeArquivo << "," << tempo << ","
              << cont.comparacoes << "," << cont.trocas << "\n";
        saida.close();
    } else {
        std::cerr << "Erro ao abrir o arquivo resultados.csv para escrita.\n";
    }
}
```

Figura 9 – Código do método medirTempo

4.3 Métodos de Busca

Para os métodos de busca, foi implementado um cabeçalho responsável por declarar as funções da Busca Sequencial e Busca Binária, bem como suas respectivas versões com medição de tempo de execução e contagem de comparações. Além disso, o cabeçalho também inclui funções auxiliares para leitura e gravação de arquivos binários, permitindo o carregamento e salvamento dos vetores utilizados nos testes.

```
#ifndef BUSCAS_H
#define BUSCAS_H


#include <vector>
#include <chrono>
#include <string>
#include <utility>


/*=====================*/
// Métodos para buscas  //
/*=====================*/

// Busca Sequencial (básica)
template <typename T>
int busca_sequencial(const std::vector<T>& vetor, const T& alvo, int& comparacoes);

// Busca Binária (requer vetor ordenado)
template <typename T>
int busca_binaria(const std::vector<T>& vetor, const T& alvo, int& comparacoes);

// Busca Sequencial com medição de tempo
template <typename T>
std::tuple<int, double, int> executar_busca_sequencial(const std::vector<T>& vetor, const T& alvo);

// Busca Binária com medição de tempo
template <typename T>
std::tuple<int, double, int> executar_busca_binaria(const std::vector<T>& vetor, const T& alvo);


/*=====================*/
// Métodos para arquivos//
/*=====================*/

// Lê um arquivo binário de floats e retorna um vector
std::vector<float> ler_arquivo(const std::string& caminho);

// Salva um vector de floats em um arquivo binário
void salvar_arquivo(const std::string& caminho, const std::vector<float>& dados);

#endif
```

Figura 10 – Código do cabeçalho dos métodos de busca

A função executar_busca_sequencial, hbcomo seu próprio nome diz, realiza a execução de uma busca sequencial em um vetor genérico, medindo seu desempenho. Ela começa registrando o tempo de início da operação com a biblioteca <chrono>. Em seguida, chama a função busca_sequencial, passando o vetor, o valor-alvo e uma variável de contagem de comparações, que é atualizada durante a busca. Ao final, registra o tempo de término, calcula a duração total da operação em milissegundos e retorna uma tupla contendo a posição onde o valor foi encontrado, o tempo de execução e o número de comparações realizadas.

Já a função executar_busca_binaria segue uma estrutura semelhante, mas utiliza

a técnica de busca binária, que pressupõe que o vetor esteja ordenado. Assim como na busca sequencial, ela registra o tempo de início e fim da operação, conta o número de comparações feitas e retorna esses dados organizados em uma tupla. Por ser mais eficiente em vetores ordenados, essa função geralmente apresenta menor tempo de execução e número de comparações, o que permite uma análise comparativa clara entre os dois métodos.

```cpp
/*====================================*/
/*         Busca Sequencial           */
/*====================================*/

template <typename T>
std::tuple<int, double, int> executar_busca_sequencial(const std::vector<T>& vetor, const T& alvo) {
    auto inicio = std::chrono::high_resolution_clock::now();
    int comparacoes = 0;
    int posicao = busca_sequencial(vetor, alvo, comparacoes);
    auto fim = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double, std::milli> duracao = fim - inicio;
    return {posicao, duracao.count(), comparacoes};
}

/*====================================*/
/*            Busca Binária           */
/*====================================*/

template <typename T>
std::tuple<int, double, int> executar_busca_binaria(const std::vector<T>& vetor, const T& alvo) {
    auto inicio = std::chrono::high_resolution_clock::now();
    int comparacoes = 0;
    int posicao = busca_binaria(vetor, alvo, comparacoes);
    auto fim = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double, std::milli> duracao = fim - inicio;
    return {posicao, duracao.count(), comparacoes};
}



/*====================================*/
// Instanciação explícita para float   //(para o linker encontrar as implementações e não retornar referência indefinida)
/*====================================*/

template int busca_sequencial<float>(const std::vector<float>&, const float&, int&);
template int busca_binaria<float>(const std::vector<float>&, const float&, int&);
template std::tuple<int, double, int> executar_busca_sequencial<float>(const std::vector<float>&, const float&);
template std::tuple<int, double, int> executar_busca_binaria<float>(const std::vector<float>&, const float&);


```

Figura 11 – Código das funções de busca

A função ler_arquivo é responsável por ler dados de um arquivo binário e armazená-los em um vetor de números do tipo float. Ela recebe como parâmetro uma string com o caminho do arquivo e retorna um vetor de floats contendo os dados lidos.

Primeiro, o arquivo é aberto em modo binário e caso a abertura falhe, a função exibe uma mensagem de erro detalhada. Se o arquivo for aberto com sucesso, a função determina seu tamanho total em bytes movendo o ponteiro de leitura para o final e obtem a posição atual, que corresponde ao tamanho do arquivo. Em seguida, o ponteiro volta para o início.

Sabendo o tamanho do arquivo em bytes, a função calcula quantos valores floats estão armazenados ali, dividindo o total de bytes pelo tamanho de um float. Depois, um vetor é criado com esse número de elementos. Em seguida, a função realiza a leitura dos dados binários diretamente para a memória do vetor.

Após a leitura, é feita uma verificação para garantir que todos os dados foram

lidos corretamente. Se ocorrer algum problema durante a leitura, uma mensagem de erro é exibida e o programa é encerrado. Por fim, o arquivo é fechado e o vetor com os dados lidos é retornado.

```
88  /*====================================*/
89  /*            Leitor Arquivo          */
90  /*====================================*/
91
92  std::vector<float> ler_arquivo(const std::string& caminho) {
93      // Abre o arquivo em modo binário
94      std::ifstream arquivo(caminho, std::ios::binary);
95
96      // Verifica se o arquivo foi aberto corretamente
97      if (!arquivo.is_open()) {
98          std::cerr << "Erro ao abrir o arquivo '" << caminho << "': "
99                    << std::strerror(errno) << std::endl;
100         exit(1); // Encerra o programa com erro
101     }
102
103     // Obtém o tamanho do arquivo
104     arquivo.seekg(0, std::ios::end); // Vai para o final
105     std::streampos tamanho = arquivo.tellg(); // Obtém a posição (tamanho)
106     arquivo.seekg(0, std::ios::beg); // Volta para o início
107
108     // Calcula quantos floats existem no arquivo
109     size_t num_floats = tamanho / sizeof(float);
110     std::vector<float> dados(num_floats); // Cria vetor com tamanho calculado
111
112     // Lê os dados binários diretamente para o vetor
113     arquivo.read(reinterpret_cast<char*>(dados.data()), tamanho);
114
115     // Verifica se a leitura foi completa
116     if (!arquivo) {
117         std::cerr << "Erro ao ler o arquivo '" << caminho << "': "
118                   << std::strerror(errno) << std::endl;
119         exit(1);
120     }
121
122     arquivo.close();
123     return dados;
124 }
125
```

Figura 12 – Código do método de ler arquivo

A função salvar_arquivo tem como objetivo salvar os dados de um vetor de floats em um arquivo binário. Ela recebe dois parâmetros: caminho e vetor de dados, que contém os valores que serão escritos no arquivo. Inicialmente, o arquivo é aberto em modo de saída binária. Se houver qualquer erro na abertura, a função exibe uma mensagem de erro detalhada.

Depois de abrir corretamente o arquivo, a função escreve o conteúdo do vetor diretamente no arquivo. Isso é feito utilizando um ponteiro para os dados do vetor e o número de bytes a serem escritos, que corresponde ao tamanho do vetor multiplicado pelo tamanho de um float.

Logo após a tentativa de escrita, há uma verificação para garantir que os dados foram gravados corretamente. Se a escrita falhar, a função novamente exibe uma mensagem de erro e encerra o programa, e por fim, o arquivo é fechado.

```cpp

/*====================================*/
/*            Salvar Arquivo          */
/*====================================*/

void salvar_arquivo(const std::string& caminho, const std::vector<float>& dados) {
    std::ofstream arquivo(caminho, std::ios::binary);
    if (!arquivo.is_open()) {
        std::cerr << "Erro ao criar o arquivo '" << caminho << "': "
                  << std::strerror(errno) << std::endl;
        exit(1);
    }

    // Escreve os dados binários diretamente do vetor
    arquivo.write(reinterpret_cast<const char*>(dados.data()),
                  dados.size() * sizeof(float));

    // Verifica se a escrita foi bem-sucedida
    if (!arquivo) {
        std::cerr << "Erro ao escrever no arquivo '" << caminho << "': "
                  << std::strerror(errno) << std::endl;
        exit(1);
    }

    arquivo.close();
}
```

Figura 13 – Código do método de salvar arquivo

4.4 Analise dos resultados

Ao fim de todas as implementações e uma carga considerável de testes, foram obtidos resultados interessantes.

No arquivo main foram implementados uma função de Menu Principal e, juntamente dele, a função principal do código, a main. A implementação da main se baseou em um switch case onde cada escolha se refere a um tipo de arquivo (pequeno, médio e grande), tal como gerar os arquivos binários novamente vide necessidade (quando os valores argumentados no gerador de dados são trocados) e uma escolha de saída, para finalizar o programa. Como o arquivo mencionado é deliberadamente grande e este documento se encontra anexado ao código fonte do projeto optamos por omitir sua visualização nesta analise e, para um entendimento mais aprofundado da função será possível visualizar toda sua implementação de forma externa.

Inicialmente, aos tempos de execuções gerais dos algoritmos para cada arquivo, temos tempos moderados para as soluções que envolvem o Selection Sort e sua otimização. Já quanto às implemntações do Bubble Sort, foi possível notar uma perda de desempenho tanto do algoritmo clássico quanto da otimização do mesmo, sendo notório o custo geral deste algoritmo para ordenações de dados. Por fim temos o Insertion Sort, sua solução clássica, este método portanto, apesar de não haver uma forma de otimizá-lo, apresentou-se consistente em seus tempos de execução, mostrando tempos similares e baixos, comparados às demais soluções.

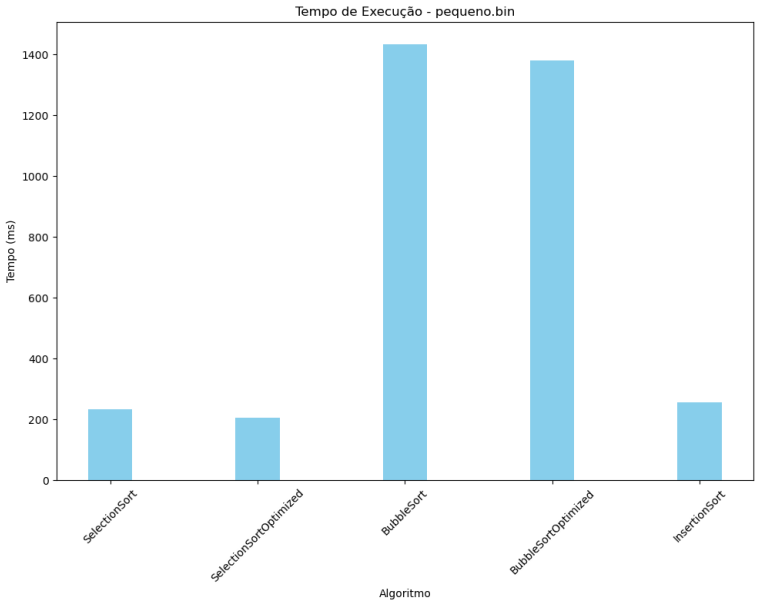

Figura 14 – Gráfico do tempo das operações no arquivo pequeno

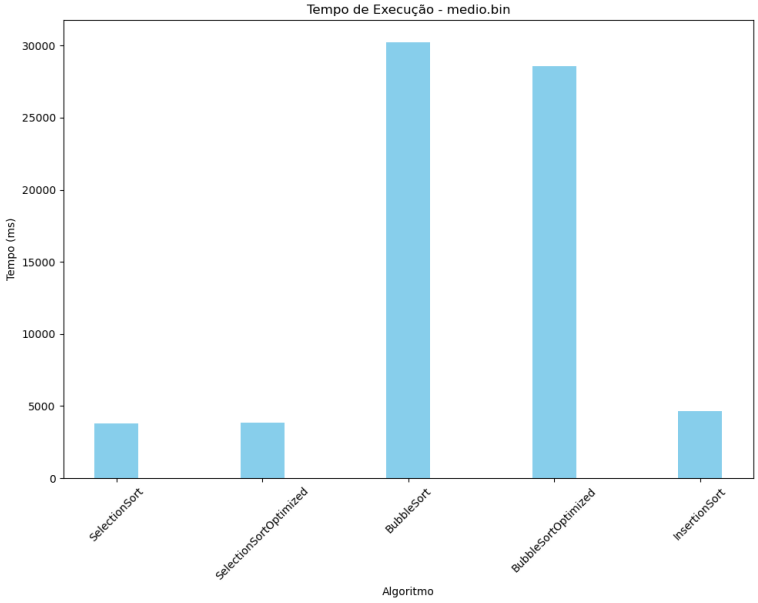

Figura 15 – Gráfico do tempo das operações no arquivo médio

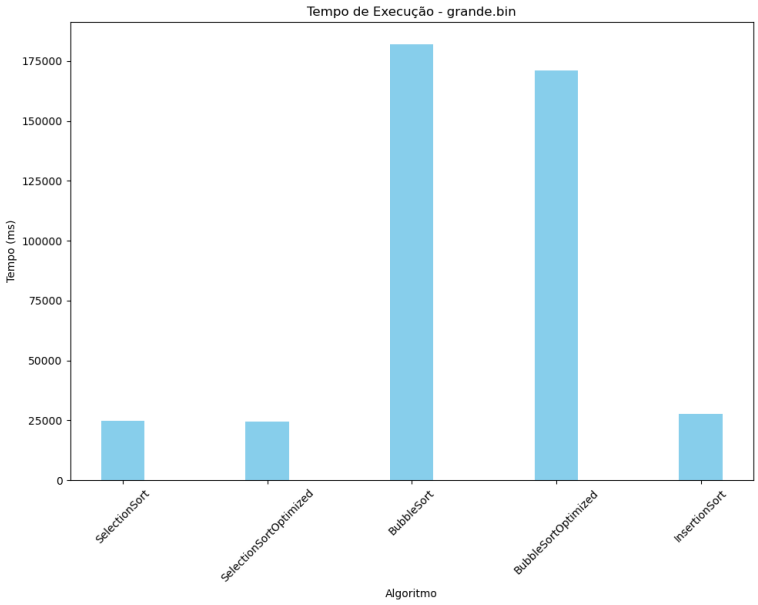

Figura 16 – Gráfico do tempo das operações no arquivo grande

Como é possível visualizar, nos algoritmos clássicos de Selection e Bubble, temos em média um desempenho inferior ao de seus aprimoramentos, mostrando a eficácia que as soluções otimizadas possuem, mesmo que não tão significativas, no contexto geral de execução.

Seguindo com os resultados, temos uma visualização clara quanto o número de comparações realizadas pelos métodos de ordenação e os respectivos arquivos binários, assim como uma amostragem das trocas realizadas por cada algoritmo.

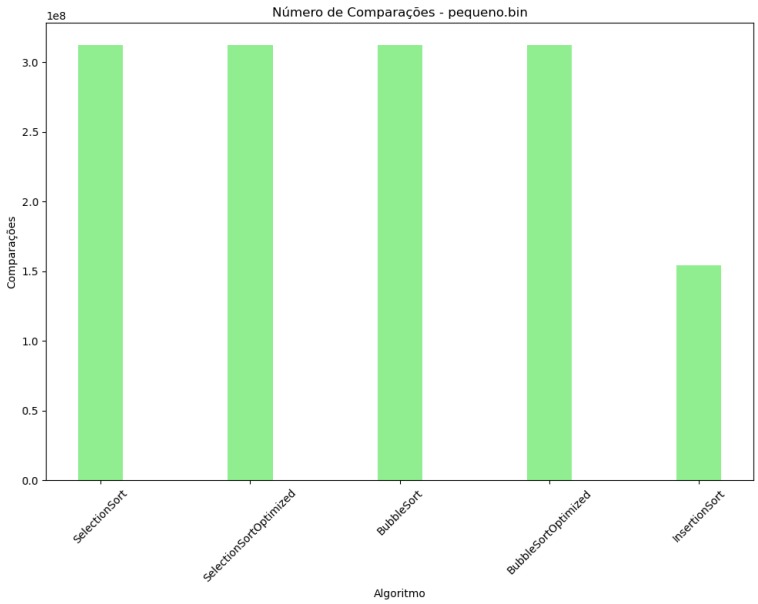

Figura 17 – Gráfico de comparações dos algoritmos no arquivo pequeno

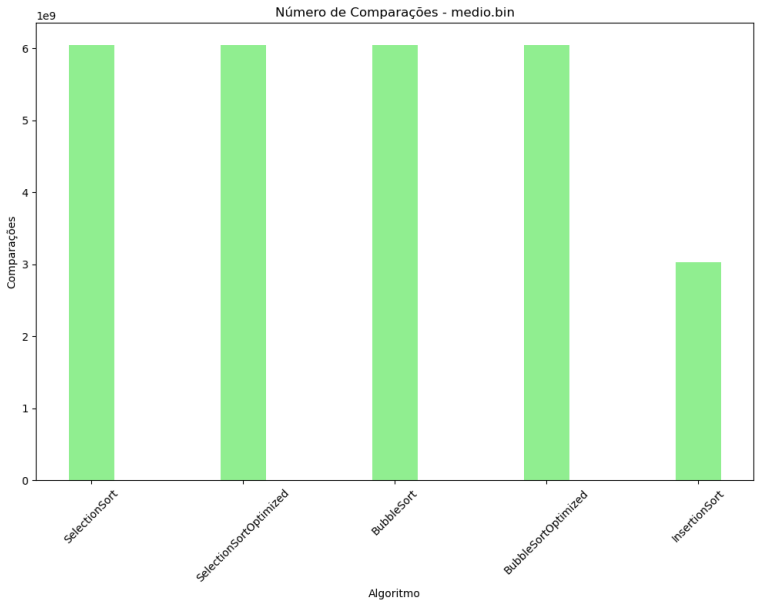

Figura 18 – Gráfico de comparações dos algoritmos no arquivo médio

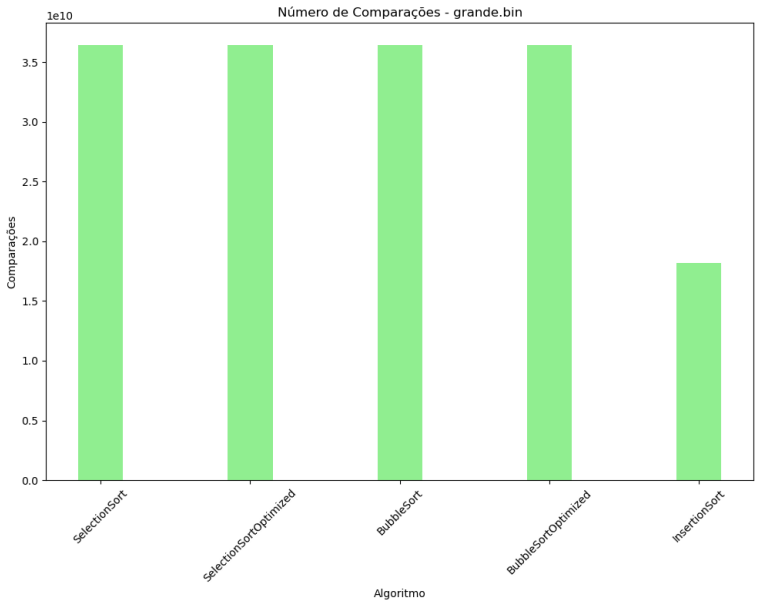

Figura 19 – Gráfico de comparações dos algoritmos no arquivo grande

Quanto às comparações dos algoritmos de ordenação, é possível notar que houveram quantidades significativas de comparações em todas as soluções implementadas, com exceção do método Insertion Sort, ele por sua vez acabou por realizar praticamente metade das comparações feitas pelos outros blocos, tornando possível perceber o quão mais otimizado este algoritmo é em relação aos demais, pois acaba rodando muito menos comparações e em um tempo total médio menor.

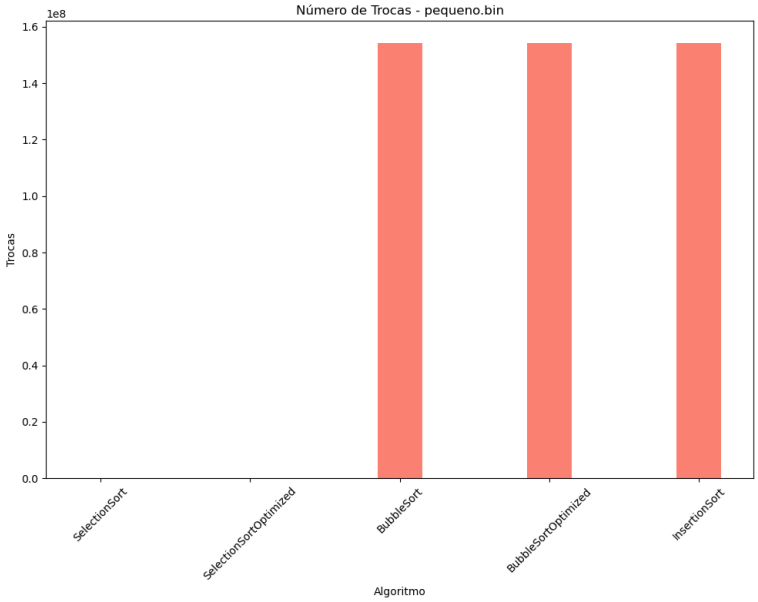

Figura 20 – Gráfico de trocas das soluções no arquivo pequeno

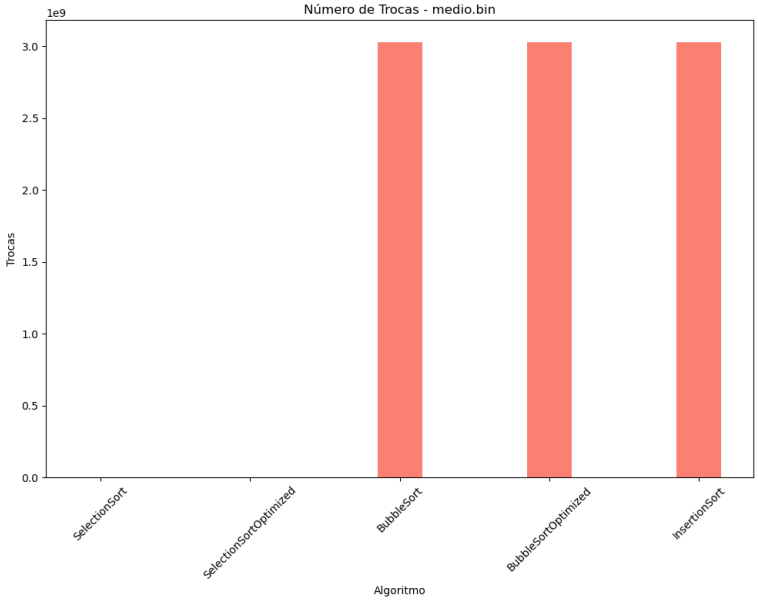

Figura 21 – Gráfico de trocas das soluções no arquivo médio

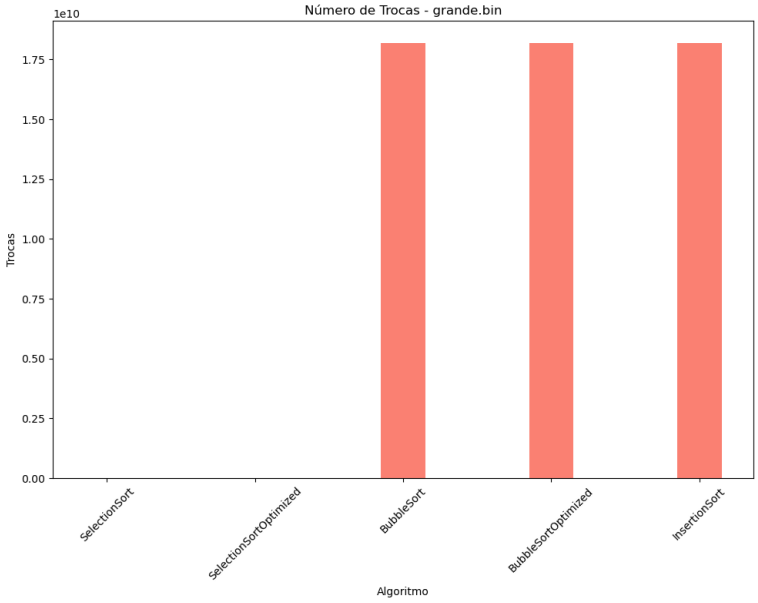

Figura 22 – Gráfico de trocas das soluções no arquivo grande

Em contrapartida, como é possível visualizar nos gráficos referentes à quantidade de trocas, temos uma quantidade relevante de trocas feitas pelos algoritmos Bubble Sort, Bubble Sort Otimizado e Insertion Sort. No entanto, ao olharmos para o Selection Sort e sua otimização, vamos que o número aparenta ser zero, porém, tal visualização se dá pelo fato de a quantidade de trocas realizadas pelos dois algoritmos ser muito pequena em comparação com as demais, deixando os gráficos com uma disparidade visível neste contexto.

Tal organização mostra o quanto o Selection Sort e sua versão otimizada são eficientes em minimizar o número de trocas, mesmo em conjuntos de dados grandes. Essa característica é particularmente vantajosa em cenários onde operações de troca são custosas, como em sistemas com restrições de memória ou dispositivos embarcados. Por outro lado, o Insertion Sort, apesar de realizar mais trocas, compensa com um número significativamente menor de comparações, destacando sua eficiência em vetores parcialmente ordenados.

Do mesmo modo temos as comparações referentes às buscas binárisa e sequenciais, nos contextos de tempos de execução e quantidade de comparações feitas.

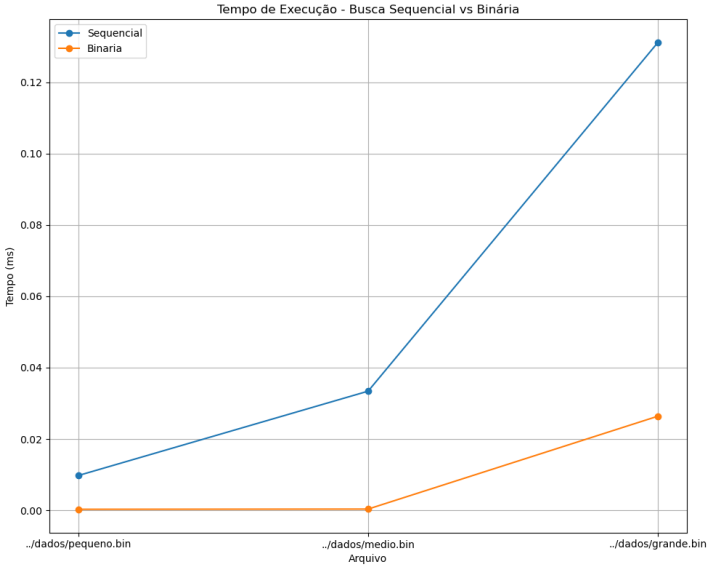

Figura 23 – Gráfico de tempo entre Busca Sequencial e Busca Binária

Os resultados demonstram uma diferença marcante entre a Busca Sequencial e a Busca Binária. Enquanto a primeira requer, em média, um número linear de comparações (O(n)), a segunda opera em tempo logarítmico (O(log n)), tornando-a drasticamente mais eficiente para conjuntos de dados ordenados. Essa vantagem é claramente refletida nos gráficos de tempo de execução, onde a Busca Binária apresenta tempos insignificantes mesmo para o arquivo grande, contrastando com os tempos crescentes da Busca Sequencial.

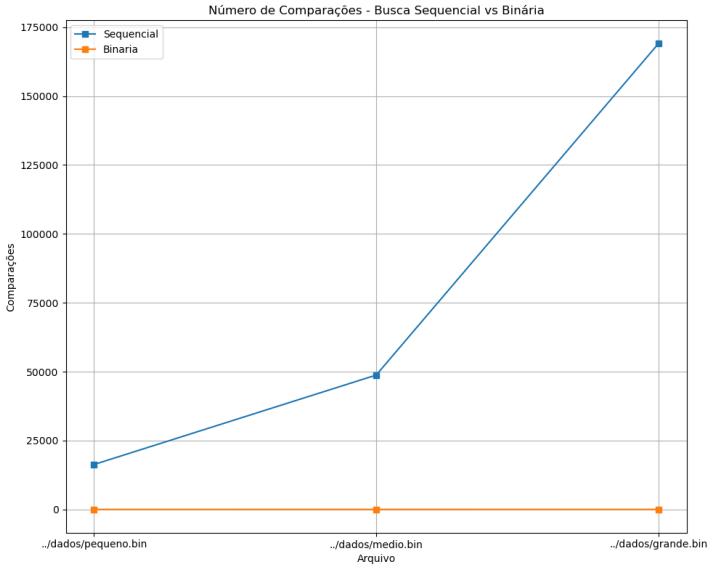

Figura 24 – Gráfico das quantidades de comparações feitas entre cada tipo de busca

A Figura acima demonstra de forma incontestável a vantagem da Busca Binária em cenários onde a ordenação prévia é viável. Enquanto a Busca Sequencial se torna impraticável para conjuntos de dados grandes, a Binária mantém um desempenho quase constante, destacando-se como a escolha ideal para otimização de desempenho em aplicações críticas. A decisão entre os dois métodos deve considerar tamanho de conjuntos de dados, a frequência de busca e o custo de manter os dados ordenados.

A Busca Sequencial opera verificando cada elemento do vetor um por um até encontrar o valor desejado. Esse método possui complexidade linear (O(n)), o que significa que o número de comparações cresce proporcionalmente ao tamanho do conjunto de dados. No pior caso (quando o elemento não está presente ou está na última posição), o algoritmo realiza exatamente 'n' comparações, onde 'n' é o número de elementos. No caso médio, considerando uma distribuição uniforme, são necessárias aproximadamente n/2 comparações. Esse crescimento linear torna a Busca Sequencial ineficiente para grandes volumes de dados, especialmente em aplicações que exigem múltiplas operações de busca.

A Busca Binária, por outro lado, exige que os dados estejam ordenados e utiliza uma estratégia de "dividir para conquistar". A cada comparação, o algoritmo descarta metade do espaço de busca, resultando em uma complexidade logarítmica (O(log n)). Isso significa que o número de comparações aumenta muito lentamente, mesmo com o crescimento exponencial do tamanho do conjunto de dados.

5 Conclusões

Este estudo realizou uma análise comparativa abrangente dos algoritmos de ordenação (SelectionSort, BubbleSort e InsertionSort, em suas versões padrão e otimizadas) e dos métodos de busca (sequencial e binária), utilizando conjuntos de dados de diferentes tamanhos para avaliar seu desempenho em termos de tempo de execução, número de comparações e trocas. Os resultados obtidos permitiram extrair conclusões fundamentais sobre a eficiência e aplicabilidade de cada algoritmo, além de destacar as vantagens das otimizações implementadas.

No contexto dos algoritmos de ordenação, observou-se que o SelectionSort, tanto em sua versão clássica quanto otimizada, apresentou um desempenho moderado, com tempos de execução consistentes, porém inferiores aos do InsertionSort em conjuntos de dados maiores. Sua principal vantagem reside no baixo número de trocas realizadas, tornando-o adequado para cenários onde operações de movimentação de dados são custosas. O BubbleSort, mesmo em sua versão otimizada (que interrompe a execução caso nenhuma troca seja detectada em uma passada completa), demonstrou ser o menos eficiente entre os métodos testados, especialmente para arquivos grandes, devido à sua natureza quadrática e alta quantidade de comparações e trocas. Por outro lado, o InsertionSort destacou-se como o mais eficiente para conjuntos de dados pequenos e parcialmente ordenados, exigindo um número significativamente menor de comparações e mantendo tempos de execução competitivos mesmo em arquivos maiores. Sua simplicidade e desempenho em casos favoráveis reforçam sua utilidade em aplicações onde a entrada já apresenta algum grau de ordenação.

Quanto aos métodos de busca, os resultados foram ainda mais decisivos. A Busca Sequencial, embora simples de implementar e aplicável a conjuntos não ordenados, mostrou-se ineficiente para volumes elevados de dados, com tempos de execução e número de comparações crescendo linearmente em relação ao tamanho do arquivo. Em contraste, a Busca Binária demonstrou uma eficiência notável, reduzindo o número de comparações a um padrão logarítmico e mantendo tempos de execução mínimos mesmo para o maior arquivo testado (270.000 elementos). Essa diferença de desempenho, que chega a ser de ordens de magnitude em favor da Busca Binária, reforça a importância de pré-ordenar os dados sempre que possível, especialmente em sistemas que exigem operações de busca rápidas e repetitivas.

Além disso, as otimizações aplicadas aos algoritmos de ordenação (como a interrupção precoce no BubbleSort e a minimização de trocas no SelectionSort) comprovaram ser efetivas, ainda que em graus variados. Enquanto no BubbleSort a otimização trouxe melhorias modestas, no SelectionSort a redução de trocas desnecessárias resultou em um ganho perceptível de desempenho. Essas adaptações ressaltam a importância de entender as características específicas de cada algoritmo e ajustá-las conforme o contexto de

aplicação.

Por fim, os dados coletados e as visualizações geradas permitiram uma comparação clara e objetiva entre os métodos, destacando trade-offs como complexidade teórica versus desempenho prático, custo de operações de comparação versus trocas, e a relação entre tamanho dos dados e eficiência algorítmica. Este trabalho não apenas cumpriu os objetivos propostos, mas também forneceu insights valiosos para a seleção de algoritmos em projetos futuros, evidenciando que a escolha do método mais adequado depende fortemente do tamanho e da natureza dos dados, bem como dos requisitos de desempenho da aplicação. A combinação de análises quantitativas e qualitativas, aliada à geração de gráficos comparativos, enriqueceu a compreensão dos resultados e serviu como base para decisões embasadas em evidências.

Em síntese, os experimentos realizados confirmaram a superioridade do Insertion-Sort para ordenação em cenários específicos, a eficácia indiscutível da Busca Binária em conjuntos ordenados e o impacto positivo de otimizações bem projetadas. Esses aprendizados são essenciais para o desenvolvimento de sistemas computacionais eficientes e escaláveis, reforçando a relevância do estudo de algoritmos clássicos e suas variações na formação em Ciência da Computação.

6 Bibliografia

https://github.com/L33tSh4rk/Analysis-of-Sorting-Algorithms