Ministério da Educação
Universidade Tecnológica Federal Do Paraná
Departamento Acadêmico de Computação
Bacharelado em Ciência da Computação

CAROLINE MARQUES LAU

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO

Campo Mourão

2025

1. Introdução

Existem alguns tipos de algoritmos para ordenar dados de forma crescente, do menor para o maior. Esses algoritmos são denominados algoritmos de ordenação. Com a finalidade de implementar, comparar e analisar o desempenho de três desses, foram gerados arquivos de três tamanhos diferentes com valores desordenados e aleatórios para serem ordenados de forma crescente. O objetivo é interpretar cada um dos resultados das ordenações e compará-los. Ademais, foram implementados os dois tipos de busca, binária e sequencial, para analisar suas diferenças ao buscar um valor em um dos arquivos ordenados. Esse trabalho será dividido em sete partes: 1. Introdução, 2. Algoritmos de ordenação, 3. Algoritmos de busca. 4. Geração de dados, 5. Análise de Desempenho, 6. Análise de desempenho e 7. Considerações finais.

2. Algoritmos de ordenação

Dentre as estratégias de ordenação, todas elas possuem n iterações, sendo n o tamanho do vetor, e duas delas possuem versões otimizadas que serão contempladas a seguir. Para obter a quantidade de comparações e trocas feitas ao utilizar cada método, cada algoritmo retorna essas variáveis.

2.1. BubbleSort

O BubbleSort caracteriza-se pela comparação de dois elementos adjacentes, caso o posterior possua um valor menor, eles são trocados. Ao fim da primeira iteração, o maior valor dentre os dados é colocado por último. A próxima iteração fará as comparações até o penúltimo elemento e colocará o segundo maior no penúltimo lugar, e assim por diante.

Código 1 - BubbleSort

```
pair<long long, long long> bubbleSort(vector<int>& vec){
        long long comp = 0;
        long long trocas = 0;
        for (size_t i = 0; i < vec.size(); i++){
                for (size_t j = 0; j < vec.size()-1-i; j++){
                        comp++;
                        if(vec[j]>vec[j+1]){
                                trocas++;
                                swap(vec[j], vec[j+1]);
                        }
```

```
                }
        }
        return {comp, trocas};
}
```

Fonte: autoria própria, 2025.

2.1.1. BubbleSortOp

A versão otimizada do BubbleSort contém uma flag que verifica se existiu alguma troca ao final de uma iteração, caso não tenha ocorrido, o código é encerrado pois o vetor já está ordenado.

Código 2 - BubbleSort otimizado

```
pair<long long, long long> bubbleSortOp(vector<int>& vec){
        long long comp = 0;
        long long trocas = 0;
        for (size_t i = 0; i < vec.size(); i++){
                bool swaped = false;
                for (size_t j = 0; j < vec.size()-1-i; j++){
                        comp++;
                        if(vec[j]>vec[j+1]){
                                trocas++;
                                swap(vec[j], vec[j+1]);
                                swaped = true;
                        }
                }
                if(!swaped) break;
        }
        return {comp, trocas};
}
```

Fonte: autoria própria, 2025.

2.2. SelectionSort

O SelectionSort, na primeira iteração, passa por todos os dados procurando o menor dentre eles e depois o coloca no começo. A segunda iteração começa a partir do segundo valor, pois o primeiro já é o menor e assim sucessivamente.

Código 3 - SelectionSort

```
pair<long long, long long> selectionSort(vector<int>& vec){
```

```
    long long comp = 0;
    long long trocas = 0;
    for (size_t i = 0; i < vec.size()-1; i++){
        int minimum = i;
        for (size_t j = i+1; j < vec.size(); j++){
            comp++;
            if(vec[minimum]>vec[j]) {
                minimum = j;
            }
        }
        if(i!=minimum){
            trocas++;
            int troca = vec[i];
            vec[i] = vec[minimum];
            vec[minimum] = troca;
        }
    }
    return {comp, trocas};
}
```

Fonte: autoria própria, 2025.

2.2.1. SelectionSortOp

A versão otimizada do SelectionSort também possui uma flag que é acionada dentro de uma comparação de dados adjacentes, se eles estiverem fora de ordem essa flag é acionada. Ao final de cada iteração, se ela não tiver sido acionada, os dados já estão ordenados e por isso a função encerra.

Código 4 - SelectionSort otimizado

```
pair<long long, long long> selectionSortOp(vector<int> &vec){
    long long comp = 0;
    long long trocas = 0;
    for (size_t i = 0; i < vec.size()-1; i++){
            bool flag = false;
        int minimum = i;
        for (size_t j = i+1; j < vec.size(); j++){
            comp++;
            if(vec[minimum]>vec[j]) {
                minimum = j;
```

```
            }
            comp++;
            if(vec[j-1]>vec[j]) flag = true;
        }
        if(!flag) break;
        if(i!=minimum){
            trocas++;
            int troca = vec[i];
            vec[i] = vec[minimum];
            vec[minimum] = troca;
        }
    }
    return {comp, trocas};
}
```

Fonte: autoria própria, 2025.

2.3. InsertionSort

O InsertionSort inicialmente considera o primeiro elemento como ordenado e o compara com o segundo, caso ele seja maior, eles trocam de lugar. Na segunda iteração, verifica-se se o terceiro elemento é menor que o segundo, se não for, o código começa a analisar o quarto elemento. No entanto, caso o terceiro elemento seja menor que o segundo, o segundo passa para o lugar do terceiro e o antigo terceiro elemento é comparado com o primeiro, caso seja maior, o primeiro passa a ser o segundo e o valor analisado é colocado no começo. Se o primeiro elemento for menor que o valor analisado, esse é colocado na segunda posição. E essa lógica continua para cada elemento do vetor.

Código 5 - InsertionSort

```
    pair<long long, long long> insertionSort(vector<int>& vec){
        long long comp = 0;
        long long trocas = 0;
        for (size_t i = 1; i < vec.size(); i++){
            int key = vec[i];
            int j = i-1;
            comp++;
            while (j >= 0 && key<vec[j]){
                comp++;
```

```
                vec[j+1] = vec[j];
                trocas++;
                j--;
            }
            trocas++;
            vec[j+1] = key;
        }
        return {comp, trocas};
    }
```

Fonte: autoria própria, 2025.

3. Algoritmos de busca

Os algoritmos de busca são responsáveis por encontrar um valor específico dentro de um vetor ordenado de forma crescente, caso esse não exista, o valor retornado é -1. Para ser possível analisar os resultados, ambas as funções retornam as trocas e comparações realizadas.

3.1. Busca sequencial

Essa estratégia de busca é caracterizada por ser uma busca linear, que vai de elemento em elemento procurando o valor especificado.

Código 6 - Busca sequencial

```
pair<int, long long> sequencialSearch(vector<int> vec, int elem){
    long long comp = 0;
    for (int i = 0; i < vec.size(); i++)
    {
        comp++;
        if(vec[i]==elem) return {i, comp};
    }
    return {-1, comp};
}
```

Fonte: autoria própria, 2025.

3.2. Busca binária

A busca binária inicialmente divide o vetor ao meio e compara se o valor buscado está à direita ou à esquerda do valor do meio. Caso esteja à direita, o intervalo passa a ser

um mais o meio e o final do vetor. Caso esteja à esquerda, o intervalo passa a ser do começo até o meio menos um. O intervalo é dividido ao meio de novo, o valor do meio é comparado com o valor a ser buscado e essa lógica continua até que o meio seja igual ao buscado ou até que o valor inicial do intervalo seja igual ao final, ou seja, quando o valor não é encontrado.

Código 7 - Busca binária

```
pair<int, long long> binarySearch(vector<int> vec, int elem){
    int start = 0;
    long long comp = 0;
    int end = vec.size()-1;
    while(start<=end){
        int half = (end-start)/2+start;
        comp++;
        if(elem>vec[half]) start = half + 1;
        else if (elem<vec[half]) end = half - 1;
        else return {half, comp};
    }
    return {-1, comp};
}
```

Fonte: autoria própria, 2025.

4. Geração de dados

Foram criados três arquivos binários para análise. Calibrados pelo tempo de ordenação do BubbleSort, o primeiro arquivo (nomeado “small_file.bin”), com o tempo de ordenação de aproximadamente um segundo, possui 11300 valores inteiros gerados aleatoriamente, resultando em um arquivo com o tamanho de 42,5 kB. O segundo (nomeado “medium_file.bin”), com o tempo de ordenação de trinta segundos, possui 59000 valores inteiros aleatórios, obtendo-se um arquivo de tamanho igual a 236 kB. Por fim, o terceiro (nomeado “large_file.bin”), com o tempo de ordenação de cento e oitenta segundos, possui 156300 valores inteiros aleatórios que resultam em um arquivo de tamanho igual a 625,2 kB.

3.1. Código gerador de dados

Para criar cada arquivo, foram utilizadas as bibliotecas do c++, “fstream” para criar e obter os arquivos e “cstdlib” para gerar os valores aleatórios.

Código 6 - Gerador de valores aleatórios em um arquivo

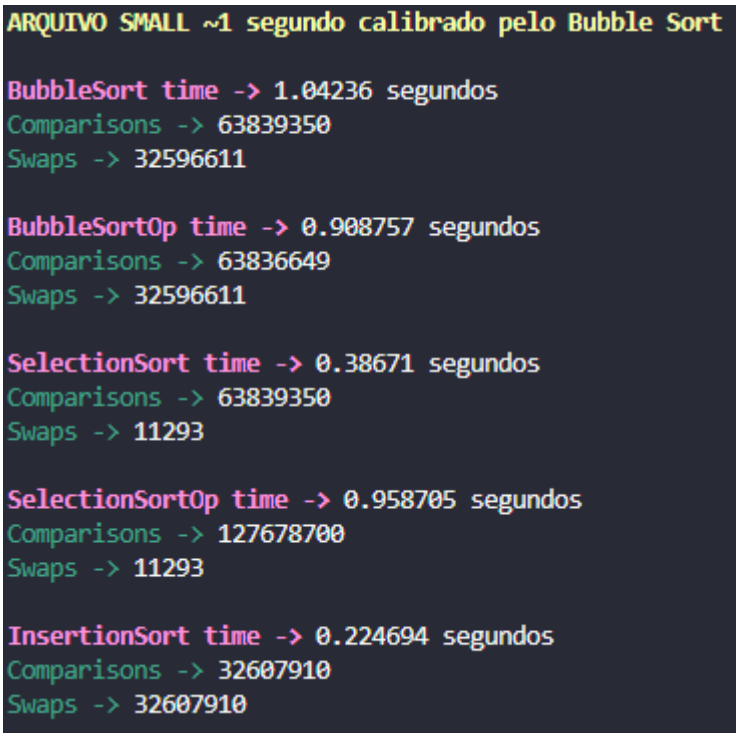
```
void writeRandomVectorInFile(string filepath, int vectorSize){
   ofstream file;
   file.open(filepath, ios::binary | ios::out);
   if(file.is_open()){
       for (int i = 0; i < vectorSize; i++)
       {
           int x = rand();
           file.write(reinterpret_cast<char*>(&x), sizeof(x));
       }
       file.close();
   } else{
       cout<<"Erro ao abrir arquivo"<< endl;
   }
}
```

Fonte: autoria própria, 2025.

5. Métodos e resultados

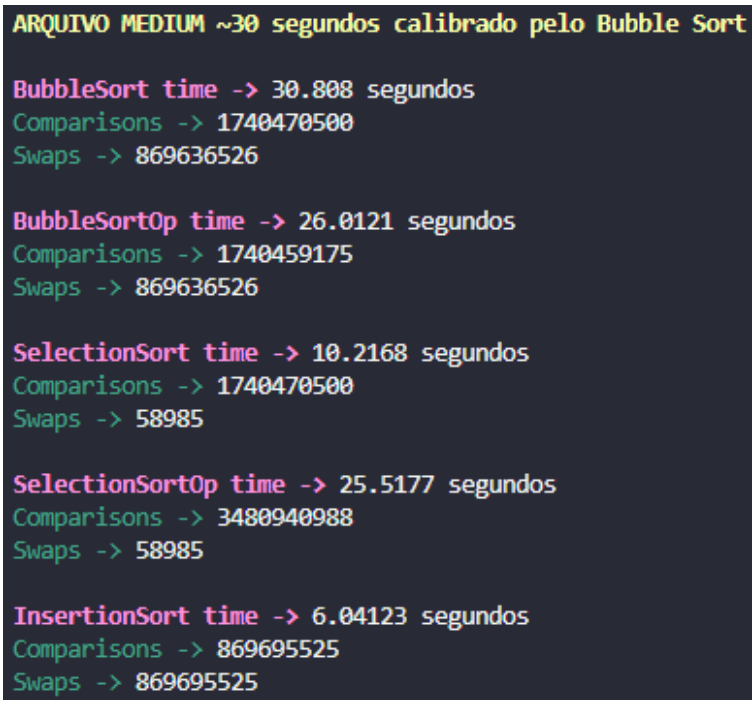
No arquivo main.cpp foram compilados cada um dos métodos de ordenação para cada tamanho de arquivo, sendo informado o tempo, por meio da biblioteca “chrono”, e número de comparações e trocas de cada um.

Figura 1 - Resultados para o arquivo pequeno



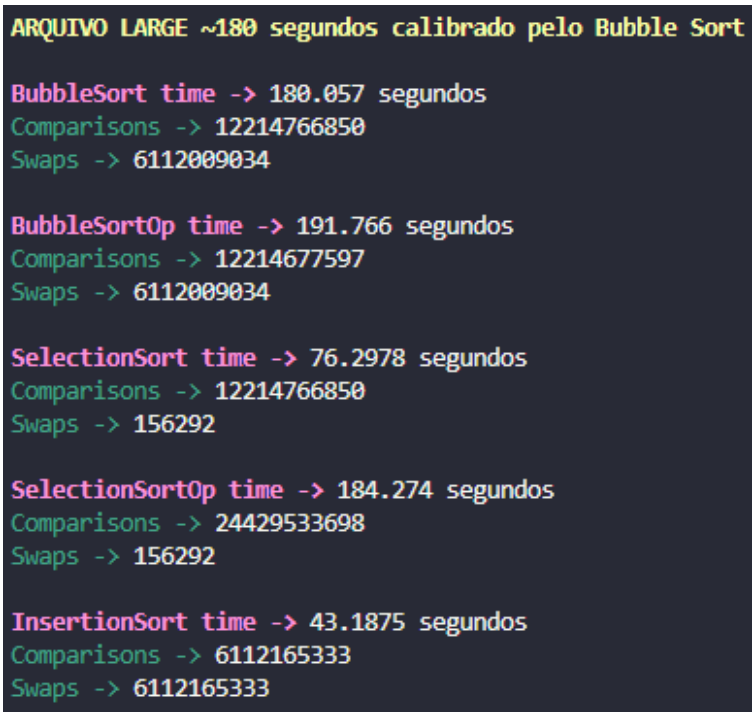
Fonte: autoria própria, 2025.

Figura 2 - Resultados para o arquivo médio



Fonte: autoria própria, 2025.

Figura 3 - Resultados para o arquivo grande



Fonte: autoria própria, 2025.

Ao final da ordenação, os códigos otimizados foram testados para o arquivo grande (Figura 4).

Figura 4 - Resultados de ordenação dos arquivos ordenados

Fonte: autoria própria, 2025.

Ademais, após a ordenação dos arquivos, as buscas foram testadas para o valor alvo 31160 (Figura 5).

Figura 4 - Resultados das buscas

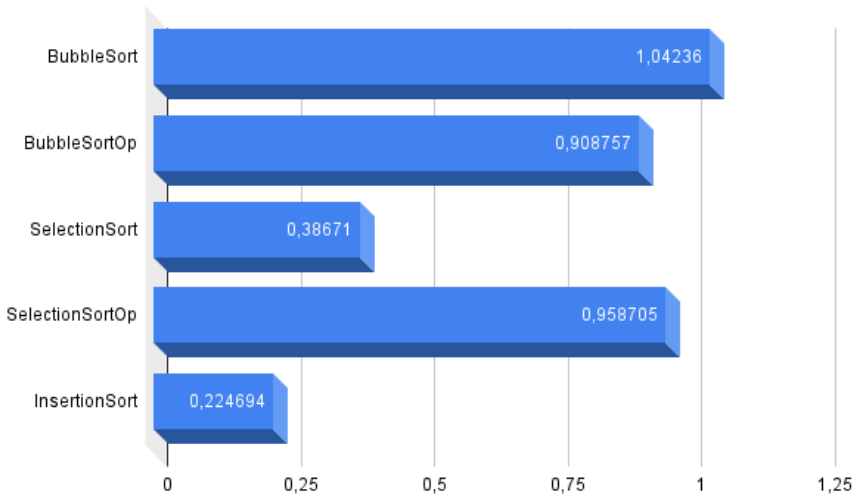
```
FIND ELEM: 31160

Binary search -> 0 segundos
Position of 31160 -> 148580
Comparisons -> 14

Sequencial search -> 0.001018 segundos
Position of 31160 -> 148580
Comparisons -> 148581
```

Fonte: autoria própria, 2025.

6. Análise de desempenho

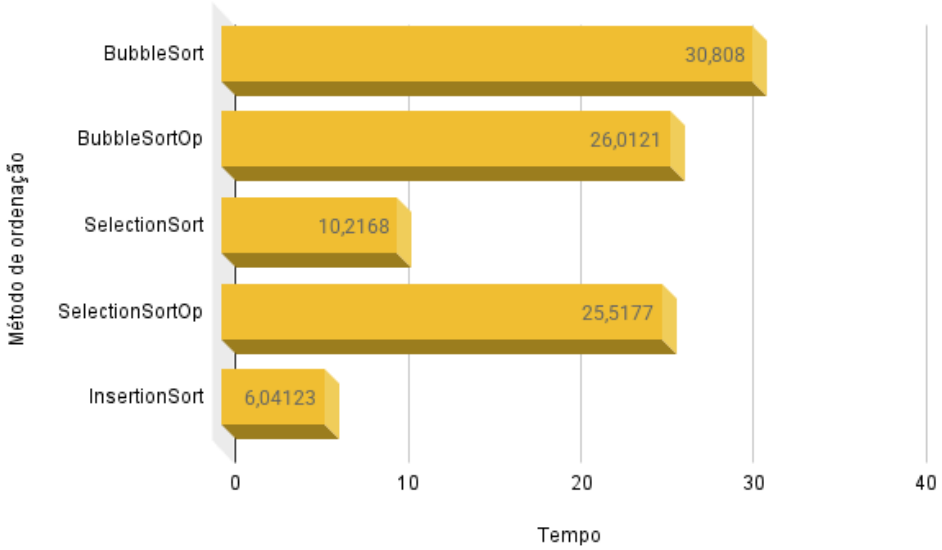
Os resultados do tempo de compilação de cada método de ordenação para cada tamanho de arquivo seguem padrões semelhantes. O método de ordenação com o menor tempo sempre é o InsertionSort. Em todos os arquivos, o tempo do InsertionSort é menos que um quarto do tempo de execução do BubbleSort (Figuras 6, 7 e 8). A razão do tempo deste método ser sempre menor é pelo fato dele ordenar aos poucos e parar de comparar assim que é encontrado um valor maior dentre os que já estão ordenados, não é necessário percorrer todos os elementos sempre.

Figura 6 - Tempo de cada método de ordenação para o arquivo pequeno



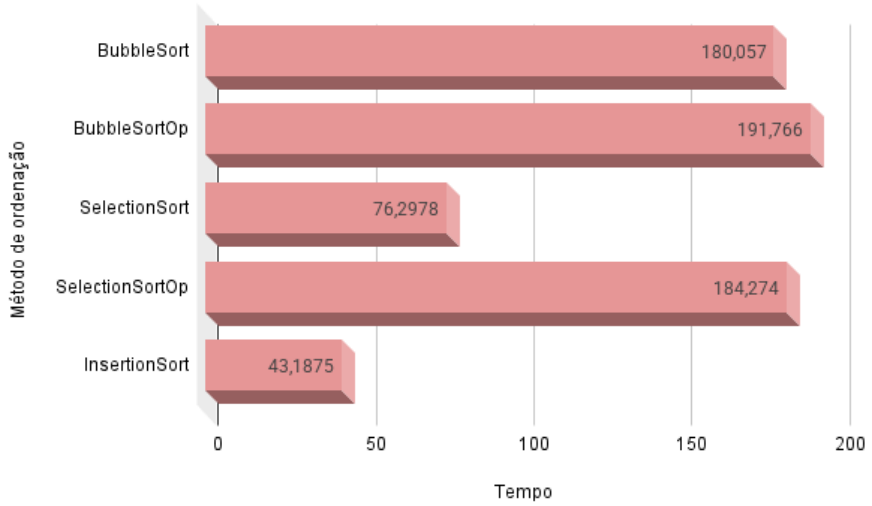
Fonte: autoria própria, 2025.

Figura 7 - Tempo de cada método de ordenação no arquivo médio



Fonte: autoria própria, 2025.

Figura 8 - Tempo de cada método de ordenação no arquivo grande



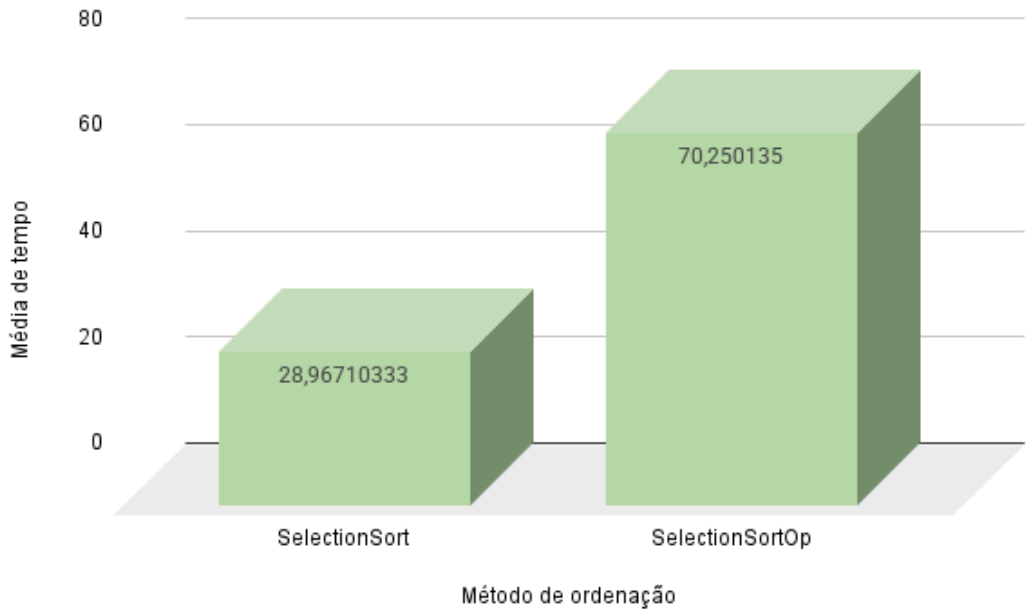
Fonte: autoria própria, 2025.

Além disso, uma vez que os códigos do BubbleSort normal e o otimizado são muito similares, os resultados de tempo em cada arquivo são sempre muito próximos. Também

vale destacar que ambos os métodos geram resultados bem discrepantes de quase todos os outros no tempo de execução em cada arquivo. Isso se deve ao fato de que a lógica do BubbleSort realiza diversas comparações em todos os casos, sempre serão comparados os valores adjacentes e as trocas vão sendo realizadas várias vezes por iteração. Em contrapartida, o SelectionSort só possui no máximo uma troca por iteração, por exemplo.

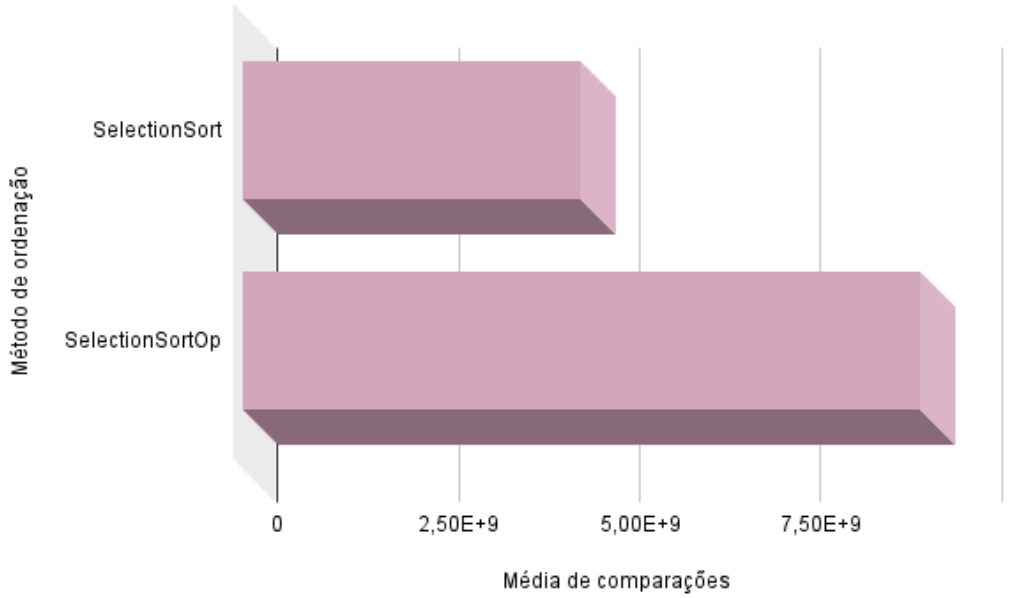
Por outro lado, os tempos do SelectionSort normal e otimizado diferem consideravelmente, em todos os casos, o código otimizado resulta em mais que o dobro do tempo do sem otimização (Figura 9). Isso é evidenciado devido ao fato da versão otimizada possuir muitas comparações a mais, já que essa compara os valores adjacentes para ver se já está ordenado (Figura 10).

Figura 9 - Média de tempo de execução do SelectionSort normal e otimizado



Fonte: autoria própria, 2025.

Figura 10 - Média de comparações do SelectionSort normal e otimizado

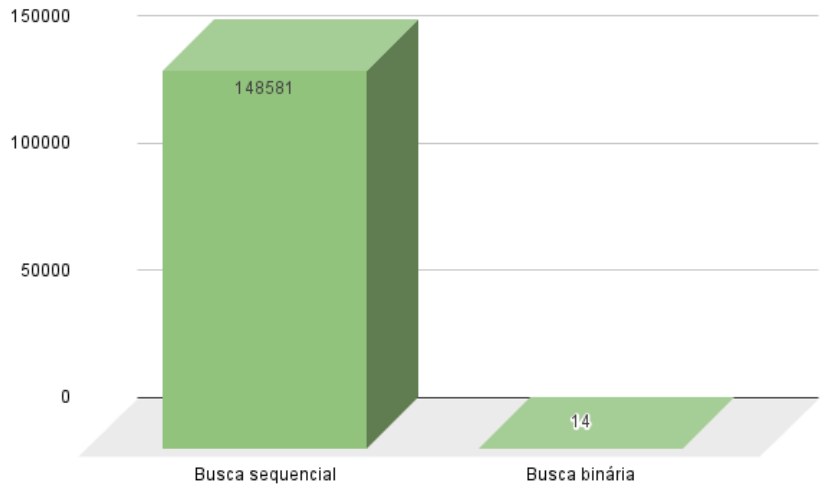


Fonte: autoria própria, 2025.

Em adição aos demais resultados, evidenciou-se que as ordenações otimizadas para os vetores ordenados foram extremamente eficazes e encerram a ordenação com um pouco mais de um milisegundo.

Além disso, tornou-se evidente que a busca sequencial é menos eficiente na grande maioria das vezes quando comparada com a sequencial, pois seu tempo é maior devido às muitas comparações (Figura 11). Por outro lado, o tempo de execução da busca binária foi tão baixo que foi considerado como 0 pelo processador. Por fim, vale ressaltar que a busca sequencial poderia ser mais eficiente quando escolhido um valor no começo da lista, mas o processador define ambos os tempos de execução como 0 para um valor no começo.

Figura 11 - Comparações nos algoritmos de busca para o valor 31160 no arquivo grande



Fonte: autoria própria, 2025.

7. Considerações finais

Conclui-se que, dentre os métodos de ordenação analisados, o mais eficiente é o InsertionSort, pois apresentou consistentemente os menores tempos de execução e número de comparações, além de não requerer uma versão otimizada para alcançar bom desempenho. O BubbleSort, em suas diferentes versões, obteve resultados semelhantes entre si, mas ainda inferiores aos do SelectionSort. Ademais, ao se considerar a execução dos métodos otimizados em vetores já ordenados, o SelectionSort demonstrou-se menos eficiente, realizando mais que o dobro de comparações em relação aos demais.