



**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ**

**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**EBER FELIPE BARROTI LOUBACK**

## **Avaliação 2 - Trabalho Prático - Algoritmos de Ordenação e Busca**

**RODRIGO HÜBNER - INTERAÇÃO HUMANO-COMPUTADOR**

**CAMPO MOURÃO**

**2025**

# Sumário:

<b>Sumário:</b> .....	<b>2</b>
<b>1. Introdução:</b> .....	<b>3</b>
<b>2. Código Fonte:</b> .....	<b>4</b>
2.1. Geração de dados:.....	4
2.2. Algoritmos de Ordenação:.....	5
2.3. Algoritmos de Busca:.....	10
2.3.1. Busca Linear:.....	10
2.3.2. Busca Binária:.....	10
2.3.3. Output: De maneira similar ao que foi feito com os algoritmos de ordenação foi utilizada uma função para exibir o tempo de execução e número de comparações feito por cada método de busca.....	11
<b>3. Análise dos Resultados:</b> .....	<b>12</b>
3.1. Algoritmos de Ordenação:.....	12
Para essa seção, os gráficos foram divididos por tamanho para auxiliar na visualização dos dados, já que manter o grupo Grande no mesmo do Pequeno resultaria no último tendo a visibilidade muito limitada, já que a maioria dos dados ficaram abaixo de 1 segundo, enquanto que os do Grande acabaram entre 50~185 segundos.....	12
3.1.1. Comparação entre os Algoritmos.....	12
3.2. Algoritmos de Busca:.....	15
<b>4. Conclusão:</b> .....	<b>17</b>
<b>5. Fontes:</b> .....	<b>18</b>

# 1. Introdução:

Este trabalho abordará uma pesquisa sobre 3 algoritmos de ordenação — *Bubble Sort (normal e otimizado para melhor caso)*, *Selection Sort (normal e otimizado para melhor caso)* e *Insertion Sort* — e 2 algoritmos de busca — *Linear Search* e *Binary Search*. Será primeiro analisado o código fonte utilizado para a análise e depois será feita a análise dos resultados obtidos, seguida pela conclusão.

Todo o código foi escrito na linguagem C++ e compilado e executado em um Lenovo IdeaPad 3 com um processador *AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz* com 8 GB de memória RAM com o computador carregando para conseguir uma melhor performance.

Código fonte do projeto disponível no GitHub pelo link: <https://github.com/enerzilius/algoritmos-ordenacao-busca.git>.

## 2. Código Fonte:

### 2.1. Geração de dados:

Todos os dados utilizados pelos algoritmos de ordenação são criados pela seguinte função:

```
void createRandomIntDataFile(int len){
    string name = "";
    if(len < 50000){
        name = "pequeno";
    }else if(len >= 50000 && len < 150000){
        name = "medio";
    }else{
        name = "grande";
    }

    string path = "dados/"+name+".bin";

    fstream file;
    file.open(path, ios::binary | ios::out);
    if(!file) return;

    for (int i = 0; i < len; i++)
    {
        int random = rand() % len;
        file.write(reinterpret_cast<char*>(&random), sizeof(int));
    }
    file.close();
}
```

Ela determina o nome do arquivo com base na quantidade de números que vão ser criados, o criando dentro da pasta *dados*, contendo a quantidade desejada de valores, que vão de 0 ao número fornecido como *len*.

Por exemplo, se a quantidade de números desejados for 14.000, serão criados 14.000 números variando entre 0 e 14.000.

## 2.2. Algoritmos de Ordenação:

Os algoritmos de ordenação utilizados foram: *Bubble Sort*, *Bubble Sort* otimizado para o melhor caso, *Selection Sort*, *Selection Sort* otimizado para o melhor caso e *Insertion Sort*. Todos eles ordenam e retornam um `vector<int>`, que é criado pela função `vectorizeData`, que lê os arquivos criados na fase de geração de dados.

```
vector<int> vectorizeData(string path){
    vector<int> empty;
    fstream file;
    file.open(path, ios::binary | ios::in);
    if(!file){
        cout<<"Falha ao abrir o arquivo no caminho indicado ["<<path<<"]\n";
        return empty;
    }

    file.seekg(0, ios::end);
    streamsize size = file.tellg();
    file.seekg(0, ios::beg);

    vector<int> data(size / sizeof(int));

    if (!file.read(reinterpret_cast<char*>(data.data()), size)) {
        std::cerr << "Erro lendo o arquivo" << std::endl;
        return empty;
    }

    file.close();
    return data;
}
```

Além da própria ordenação, todos os algoritmos também contam o número de comparações e trocas que fazem durante sua execução.

### 2.2.1. *Bubble Sort*:

O *Bubble Sort* compara pares de elementos adjacentes e os troca de posição se estiverem na ordem errada. Esse processo é repetido várias vezes, fazendo os maiores elementos irem para o final da lista a cada iteração.

```
vector<int> bubbleSort(vector<int> data, unsigned long* comps, unsigned long* switches){
    vector<int> sortedData = data;
    bool switched;

    for(int i = 0; i < sortedData.size(); i++){
        switched = false;

        for (int j = 0; j < sortedData.size()-i; j++)
        {
            *comps += 1;
            if(sortedData[j] > sortedData[j+1]){
                swap(sortedData[j], sortedData[j+1]);
                *switches += 1;
                switched = true;
            }
        }

        if(!switched) break;
    }

    return sortedData;
}
```

### 2.2.2. *Bubble Sort* Otimizado:

A diferença desse para o *Bubble Sort* convencional é que caso o número da posição atual (*i*) esteja correta ( $\leq$  o número da próxima posição) ele passa para próxima posição, sem entrar no segundo loop.

No melhor caso, o algoritmo iria de complexidade  $O(n^2)$  para  $O(n)$ .

```
vector<int> optimizedBubbleSort(vector<int> data, unsigned long* comps, unsigned long* switches){
    vector<int> sortedData = data;
    bool switched;

    for(int i = 0; i < sortedData.size(); i++){
        switched = false;
        *comps += 1;
        if(sortedData[i] <= sortedData[i+1]) continue;
        for (int j = 0; j < sortedData.size()-i; j++)
        {
            *comps += 1;
            if(sortedData[j] > sortedData[j+1]){
                swap(sortedData[j], sortedData[j+1]);
                *switches += 1;
                switched = true;
            }
        }

        if(!switched) break;
    }

    return sortedData;
}
```

### 2.2.3. Selection Sort:

O Selection Sort percorre o array e encontra o menor elemento, colocando-o na primeira posição. Em seguida, busca o segundo menor e o coloca na segunda posição, e assim por diante, até ordenar todos os números da lista.

```
vector<int> selectionSort(vector<int> data, unsigned long* comps, unsigned long* switches){
    vector<int> sortedData = data;
    int iMin = 0;

    for (int i = 0; i < sortedData.size(); i++)
    {
        iMin = i;
        for (int j = i; j < sortedData.size(); j++)
        {
            *comps += 1;
            if(sortedData[j] < sortedData[iMin]){
                iMin = j;
            }
        }
        swap(sortedData[i], sortedData[iMin]);
        *switches += 1;
    }

    return sortedData;
}
```

### 2.2.4. Selection Sort Otimizado:

Neste algoritmo foi feita a mesma coisa do *Bubble Sort* Otimizado, adicionando apenas uma validação checando se o número atual já está na posição correta, pulando para o próximo valor do vetor caso esteja.

```
vector<int> optimizedSelectionSort(vector<int> data, unsigned long* comps, unsigned long* switches){
    vector<int> sortedData = data;
    int iMin = 0;

    for (int i = 0; i < sortedData.size(); i++)
    {
        iMin = i;
        *comps += 1;
        if(sortedData[i] <= sortedData[i+1]) continue;
        for (int j = i; j < sortedData.size(); j++)
        {
            *comps += 1;
            if(sortedData[j] < sortedData[iMin]){
                iMin = j;
            }
        }
        swap(sortedData[i], sortedData[iMin]);
        *switches += 1;
    }

    return sortedData;
}
```

### 2.2.5. Insertion Sort:

O Insertion Sort divide o vetor em 2: uma parte ordenada e uma parte desordenada. A cada iteração ela pega o primeiro da parte desordenada e o coloca na posição correta na parte ordenada, fazendo isso até tudo estar ordenado.

```
vector<int> insertionSort(vector<int> data, unsigned long* comps, unsigned long* switches){
    vector<int> sortedData = data;
    int index = 0;
    int j = 0;

    for (int i = 0; i < sortedData.size(); i++){
        index = i;
        int n = sortedData[i];
        j = i-1;
        *comps += 1;
        while(j >= 0 && sortedData[j] > n){
            *comps += 1;
            *switches += 1;
            sortedData[j+1] = sortedData[j];
            index = j;
            j--;
        }
        sortedData[index] = n;
    }
    return sortedData;
}
```

### 2.2.6. Captura do Tempo de Execução e Output

Para aumentar a legibilidade do código, todo o processo de capturar o tempo de execução e imprimir seus resultados, que ocorreria várias vezes na execução do projeto foi resumido em duas funções: *createSortAnalysis* e *printAnalysis*.

Como sugerido pelo nome, uma função é responsável pela captura do tempo e a outra imprime seus resultados. Para fazer a captura do tempo de execução, foi utilizado a biblioteca *chrono*, capturando um momento no tempo antes da execução do código e um logo após e os subtraindo para obter o tempo decorrido durante o algoritmo de ordenação. Além disso, para facilitar no *output*, existem algumas variáveis de controle, decidindo o algoritmo que deve ser executado e o que deve ser imprimido no terminal.

No último algoritmo de ordenação do grupo (dividido por tamanho) é chamado o método *createBinaryFileFromVector*, que cria um arquivo binário contendo o vetor criado pelo método de ordenação chamado.



```

void printAnalysis(vector<int> data, std::chrono::duration<double> tempo, unsigned long comps, unsigned long switches, int sorting, bool otimizado){
    string obs = otimizado?" (otimizado para o melhor caso)": "";
    string algs[] = {"Bubble Sort", "Selection Sort", "Insert Sort"};
    cout<<data.size()<<" números ordenados com "<<algs[sorting-1]<<obs<<" em "<<tempo.count()<<"s com "<<comps<<" comparações e "<<switches<<" trocas\n";
}

```

```

vector<int> createSortAnalysis(int sorting, bool otimizado, vector<int> data){
    unsigned long comps = 0;
    unsigned long switches = 0;

    time_point<system_clock> t1;
    time_point<system_clock> t2;

    switch (sorting)
    {
    case 1:
        t1 = high_resolution_clock::now();
        if(otimizado) data = optimizedBubbleSort(data,&comps,&switches);
        else data = bubbleSort(data,&comps,&switches);
        t2 = high_resolution_clock::now();
        break;
    case 2:
        t1 = high_resolution_clock::now();
        if(otimizado) data = optimizedSelectionSort(data,&comps,&switches);
        else data = selectionSort(data,&comps,&switches);
        t2 = high_resolution_clock::now();
        break;
    case 3:
        t1 = high_resolution_clock::now();
        data = insertionSort(data,&comps,&switches);
        t2 = high_resolution_clock::now();
        break;
    default:
        break;
    }

    std::chrono::duration<double> tempo = t2 - t1;
    printAnalysis(data, tempo, comps, switches, sorting, otimizado);
    return data;
}

```

```

void createBinaryFileFromVector(vector<int> sorted){
    string name = "";
    if(sorted.size() < 50000){
        name = "Pequeno";
    }else if(sorted.size() >= 50000 && sorted.size() < 150000){
        name = "Medio";
    }else{
        name = "Grande";
    }

    string path = "dados/result"+name+".bin";

    fstream file;
    file.open(path, ios::binary | ios::out);
    if(!file) {
        cout<<"Falha na criação de arquivo com os resultados.\n";
        return;
    }

    file.write(reinterpret_cast<char*>(sorted.data()), sorted.size()*sizeof(int));
    file.close();
}

```

## 2.3. Algoritmos de Busca:

### 2.3.1. Busca Linear:

Passa por todos os elementos do vetor e checa um por um se é o número desejado.

```
int linear(vector<int> data, int target, unsigned long *comps){
    for(int i = 0; i < data.size(); i++){
        *comps += 1;
        if(data[i] == target){
            return i;
        }
    }
    return -1;
}
```

### 2.3.2. Busca Binária:

O algoritmo apenas funciona em vetores ordenados e divide o vetor no meio a cada iteração, checando se o valor da posição atual é menor ou maior que o elemento desejado, então reduz novamente pela metade o campo de busca e assim sucessivamente.

```
int binary(vector<int> data, int target, unsigned long *comps){
    int high = data.size()-1;
    int low = 1;

    *comps += 1;
    while(low <= high){
        int mid = (low+(high - low)/ 2);

        *comps += 1;
        if(data[mid] == target) return mid;
        *comps += 1;
        if(data[mid] < target) low = mid + 1;
        else high = mid - 1;
    }

    return -1;
}
```

### 2.3.3. Output:

De maneira similar ao que foi feito com os algoritmos de ordenação foi utilizada uma função para exibir o tempo de execução e número de comparações feito por cada método de busca.

```
void printSearchAnalysis(vector<int> data, int searchType, int size, int randomValue){
    string sizes[] = {"Pequeno", "Medio", "Grande"};
    string types[] = {"Linear Search", "Binary Search"};

    unsigned long comps = 0;

    int index = 0;

    time_point<system_clock> t1;
    time_point<system_clock> t2;

    switch(searchType){
        case 1:
            t1 = high_resolution_clock::now();
            index = linear(data, randomValue, &comps);
            t2 = high_resolution_clock::now();
            break;
        case 2:
            t1 = high_resolution_clock::now();
            index = binary(data, randomValue, &comps);
            t2 = high_resolution_clock::now();
            break;
    }
    std::chrono::duration<double> tempo = t2 - t1;

    cout<<types[searchType-1]<<" encontrou o número "<<randomValue<<" em "<<tempo.count()<<"s na posição "<<index<<" com "<<comps<<" comparações\n";
}
```

### 3. Análise dos Resultados:

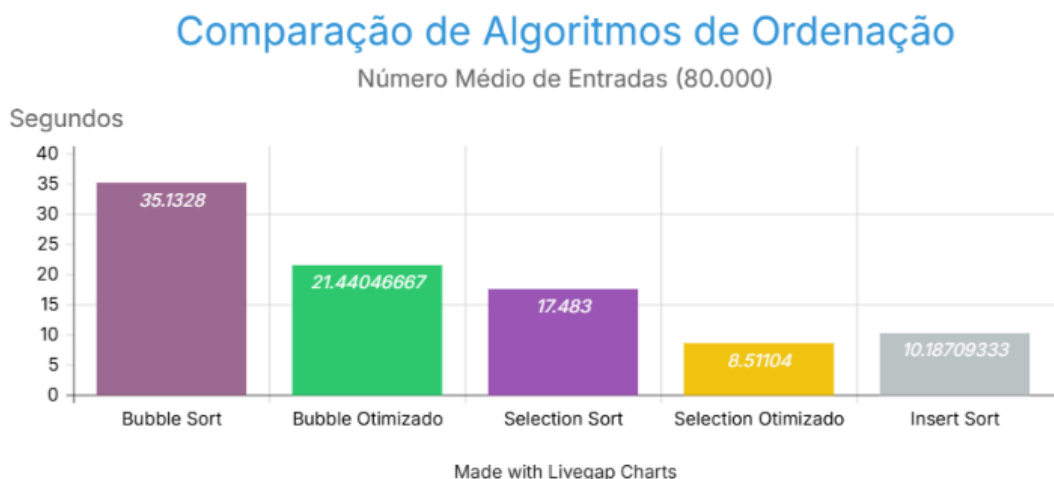
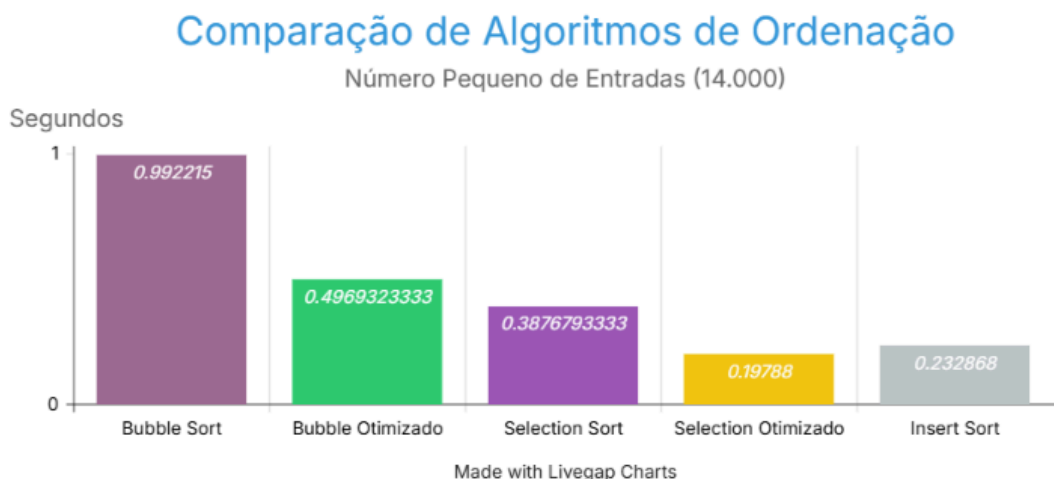
Os resultados foram alcançados analisando a média do tempo de execução de três execuções de seus respectivos códigos.

#### 3.1. Algoritmos de Ordenação:

Para essa seção, os gráficos foram divididos por tamanho para auxiliar na visualização dos dados, já que manter o grupo *Grande* no mesmo do *Pequeno* resultaria no último tendo a visibilidade muito limitada, já que a maioria dos dados ficaram abaixo de 1 segundo, enquanto que os do *Grande* acabaram entre 50~185 segundos.

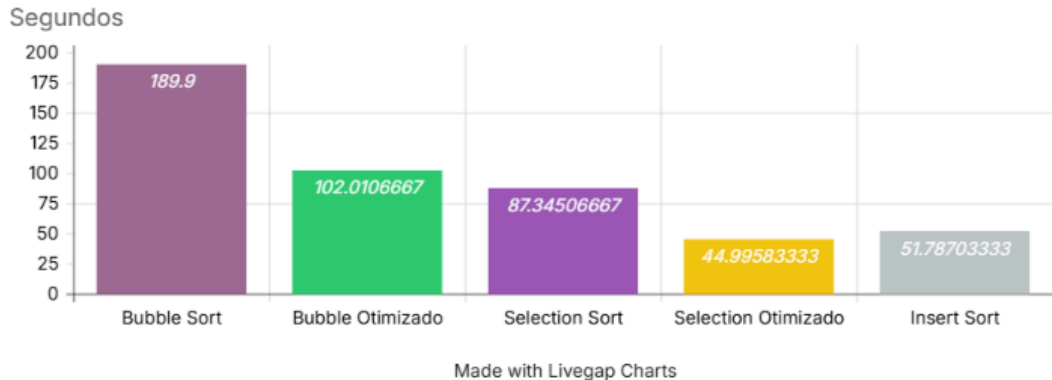
##### 3.1.1. Comparação entre os Algoritmos

###### 3.1.1.1. Casos Normais



## Comparação de Algoritmos de Ordenação

Número Grande de Entradas (180.000)



À primeira vista, é fácil dizer quais são os algoritmos mais rápidos. O *Bubble Sort* aparece como consideravelmente mais devagar que todos os outros algoritmos em todos os testes, com sua versão otimizada se mostrando entre 33% e 50% mais rápida do que ela, mesmo nos testes normais.

O *Selection Sort* se mostrou levemente mais rápido que o *Bubble Sort* Otimizado, sendo um pouco menos que 20% mais eficiente. No entanto, sua versão otimizada teve uma queda de ~50% do tempo de execução, se mostrando superior até que o *Insert Sort*, que é de longe o mais rápido de todas as versões convencionais.

O *Selection Sort* Otimizado conseguiu ser o mais rápido devido ao seu número muito reduzido de trocas, realizando em média muito menos trocas do que qualquer outro algoritmo, já que essa é a ação presente nesses algoritmos que mais consome tempo de execução.

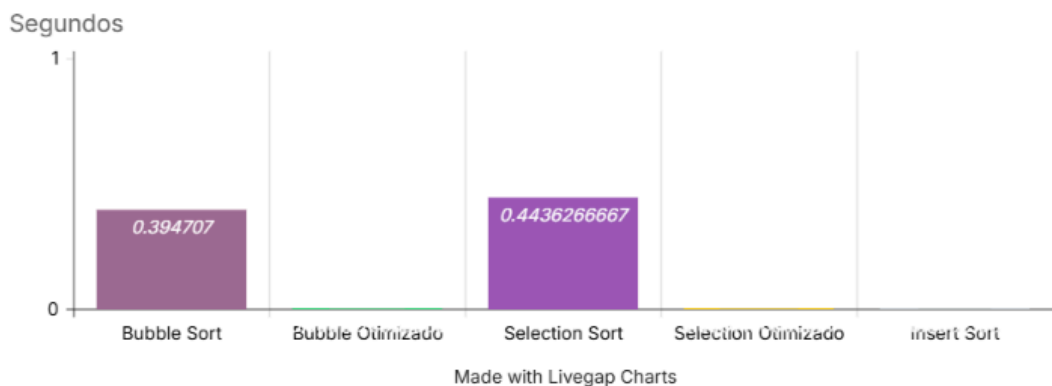
Uma análise de todos os gráficos mostra que a proporção de tempo utilizado por cada um se manteve similar entre os três casos, com todos os gráficos sendo muito similares apesar dos números diferentes. Isso evidencia que o tempo de execução de todos cresceu de forma equiparável, com nenhum tendo uma clara vantagem em casos com maiores quantidades de números, como será o caso com os algoritmos de busca.

### 3.1.1.2. Melhores Casos

Com os melhores casos, há uma discrepância enorme entre os algoritmos otimizados e o *Insertion Sort* em relação ao *Bubble Sort* e o *Selection Sort* convencionais, já que esses dois não têm a capacidade de não entrar no segundo loop caso o número da posição atual esteja na ordem correta, algo que já pode ser visto nos Casos Normais, mas fica ainda mais exagerado nos melhores casos.

## Comparação de Algoritmos de Ordenação

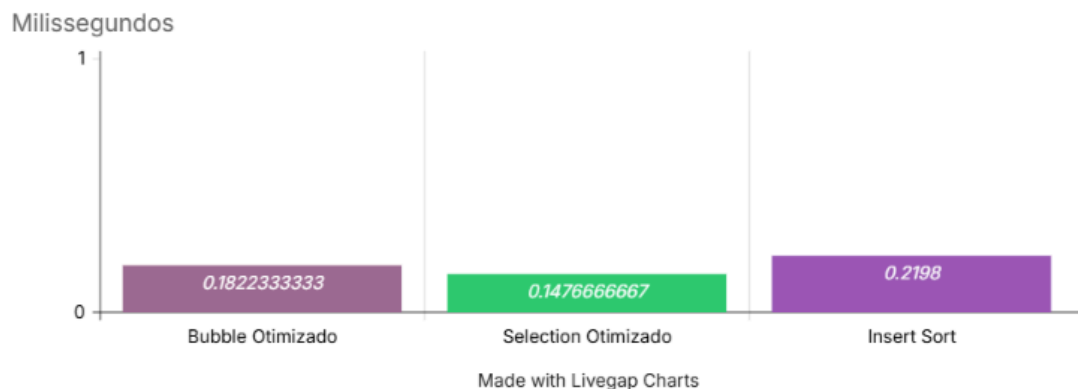
Número Pequeno de Entradas de Melhor Caso (14.000)



Por causa disso, os resultados do *Bubble Sort* e *Selection Sort* serão desconsiderados e a unidade de medida será *ms* para que seja os outros algoritmos se tornem visíveis nos gráficos.

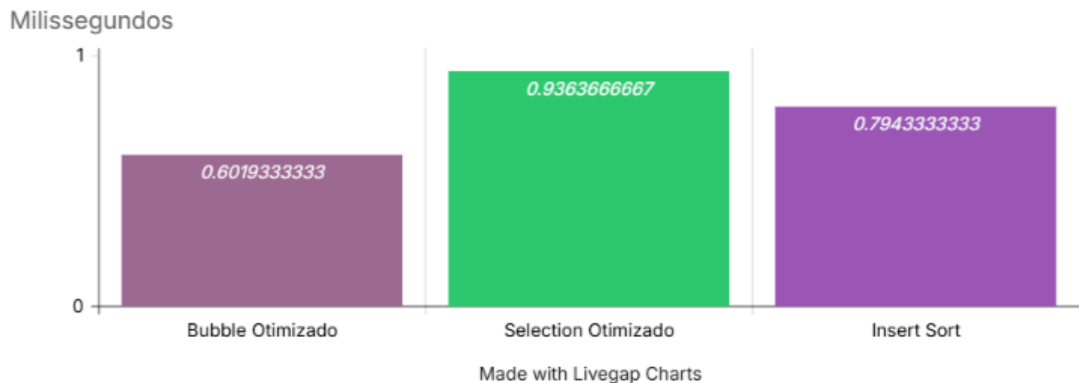
## Comparação de Algoritmos de Ordenação

Número Pequeno de Entradas de Melhor Caso (14.000)



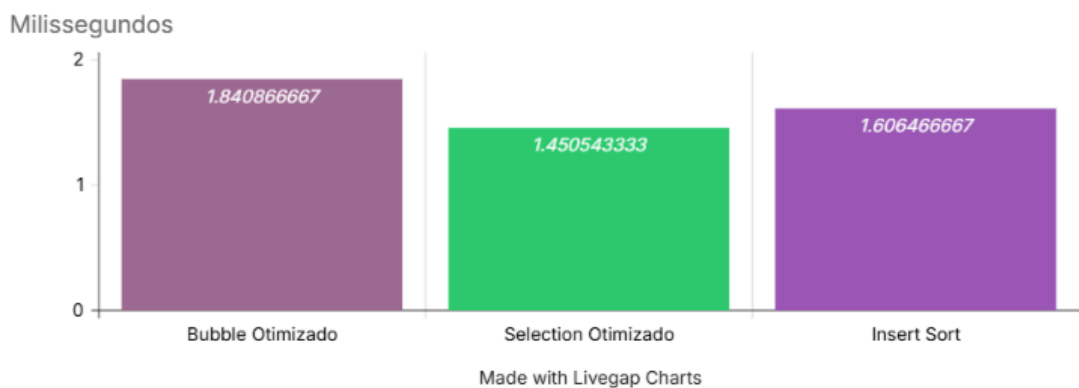
## Comparação de Algoritmos de Ordenação

Número Médio de Entradas de Melhor Caso (80.000)



## Comparação de Algoritmos de Ordenação

Número Grande de Entradas de Melhor Caso (180.000)



Neles, não há um padrão muito claro, apenas é possível ver as mesmas proporções dos casos normais no *Número Grande de Entradas*, mas no geral, os três algoritmos agem de maneiras muito similares em uma situação de melhor caso, já que todos apresentam entre 0 e 1 trocas e números muito similares de comparações nos melhores casos.

### 3.2. Algoritmos de Busca:

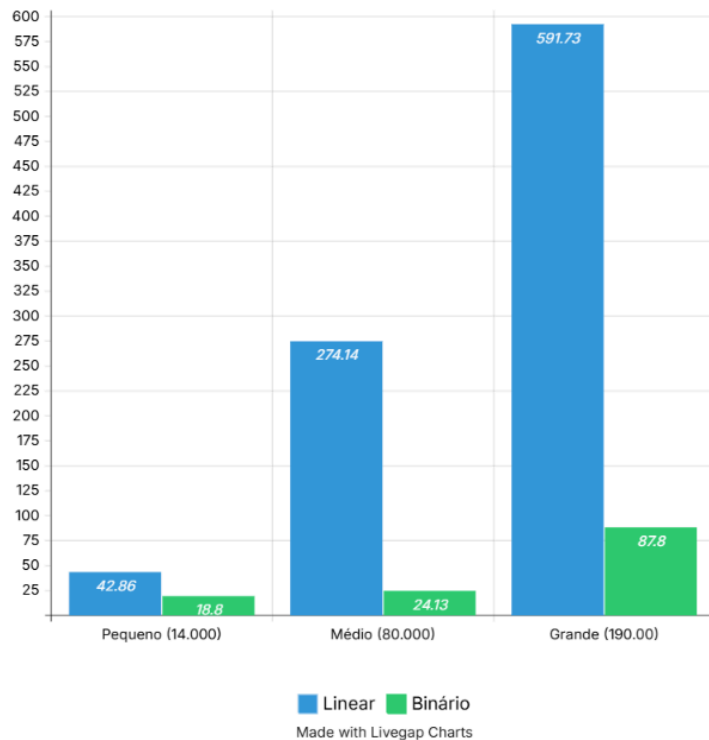
A comparação entre os algoritmos de busca foi feita em microssegundos, já que o resultado em milissegundos ou apenas segundos seria muito pequeno para uma visualização boa dos dados.

Dessa forma, a diferença de eficiência entre os dois algoritmos fica ainda mais evidente, mostrando que conforme o número de entradas aumenta, o tempo de execução da Busca Linear aumenta de forma exponencial, assim como esperado de sua complexidade  $O(n^2)$ , e que a binária sofre muito menos com o aumento da quantidade de números no vetor devido à sua complexidade  $O(\log n)$ .

Esse gráfico apenas omite o fato de mesmo que a Busca Linear – em comparação com a Busca Binária – demore muito mais, o tempo ainda é muito pequeno, tendo resultados satisfatórios na maioria das suas implementações, ainda sendo uma opção viável em sistemas com um número não tão alto de entradas, especialmente se os dados não forem previamente tratados.

### Comparação entre Busca Linear e Binária

Microssegundos





## 4. Conclusão:

Este trabalho introduziu os conceitos de alguns dos mais conhecidos algoritmos de ordenação e busca, analisando os resultados obtidos pela sua execução demonstrando na prática o conceito de complexidade do algoritmo e criando gráficos para a visualização desses conceitos.

No fim, os dados mostraram um melhor desempenho do *Selection Sort* otimizado e do *Insertion Sort* sobre os outros algoritmos de ordenação, isso se dando pelo fato de evitarem trocas e laços de repetição desnecessários. As informações obtidas também serviram para mostrar como os algoritmo de complexidade  $O(\log n)$  e  $O(n^2)$ , como a Busca Binária e a Busca Linear se comportam de forma aplicada.

## 5. Fontes:

*Binary Search*. Disponível em:

<https://www.programiz.com/dsa/binary-search>. Acesso em: 30 de maio de 2025.

*Bubble Sort Algorithm*. Disponível em:

<https://www.geeksforgeeks.org/bubble-sort-algorithm/>. Acesso em: 29 de maio de 2025.

*Insertion Sort Algorithm*. Disponível em:

<https://www.geeksforgeeks.org/insertion-sort-algorithm/>. Acesso em: 29 de maio de 2025.

*DSA Insertion Sort Algorithm*. Disponível em:

[https://www.w3schools.com/dsa/dsa\\_algo\\_insertionsort.php](https://www.w3schools.com/dsa/dsa_algo_insertionsort.php). Acesso em: 29 de maio de 2025.

*Selection Sort*. Disponível em:

<https://www.programiz.com/dsa/selection-sort>. Acesso em: 30 de maio de 2025.