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 - ALGORITMOS E ESTRUTURAS DE DADOS 1

CAMPO MOURÃO

2025

Sumário:

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* com *Radeon Graphics 1.80 GHz* e 8 GB de memória RAM, com o computador conectado à alimentação 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, criando-o 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++){
        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;
            }
        }
    }
    return sortedData;
}
```

2.2.2. *Bubble Sort Otimizado*

A diferença desta versão do algoritmo para a convencional é a capacidade de quebrar o laço de repetição caso o vetor já esteja ordenado, determinando isso ao notar que não houve troca de posição na 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;
}
```

2.2.3. *Selection Sort*

O Selection Sort percorre o vetor 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 adicionado um *loop* que checa a cada iteração se o vetor está ordenado, e caso esteja, o algoritmo sai da repetição original e retorna o vetor.

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

    for (int i = 0; i < sortedData.size(); i++)
    {
        iMin = i;
        sorted = true;
        for(int j = i; j < sortedData.size(); j++){
            *comps += 1;
            if((sortedData[j] > sortedData[j+1] || sortedData[j] < sortedData[j-1]) && j != 0) sorted = false;
        }
        *comps += 1;
        if(sorted) break;

        for (int j = i; j < sortedData.size(); j++)
        {
            *comps += 1;
            if(sortedData[j] < sortedData[iMin]){
                iMin = j;
            }
        }
        if(iMin == i) break;
        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, ele pega o primeiro valor 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 pelos nomes, uma função é responsável pela captura do tempo e a outra imprime seus resultados. Para fazer a captura do tempo de execução, foi utilizada a biblioteca *chrono*, captura um momento no tempo antes da execução do código e um logo após e os subtrai para obter o tempo decorrido durante o algoritmo de ordenação. Além disso, para facilitar o *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;
}
```

```cpp
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.

```cpp
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 ordenadas 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.

```cpp
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.

```cpp
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";
}
```

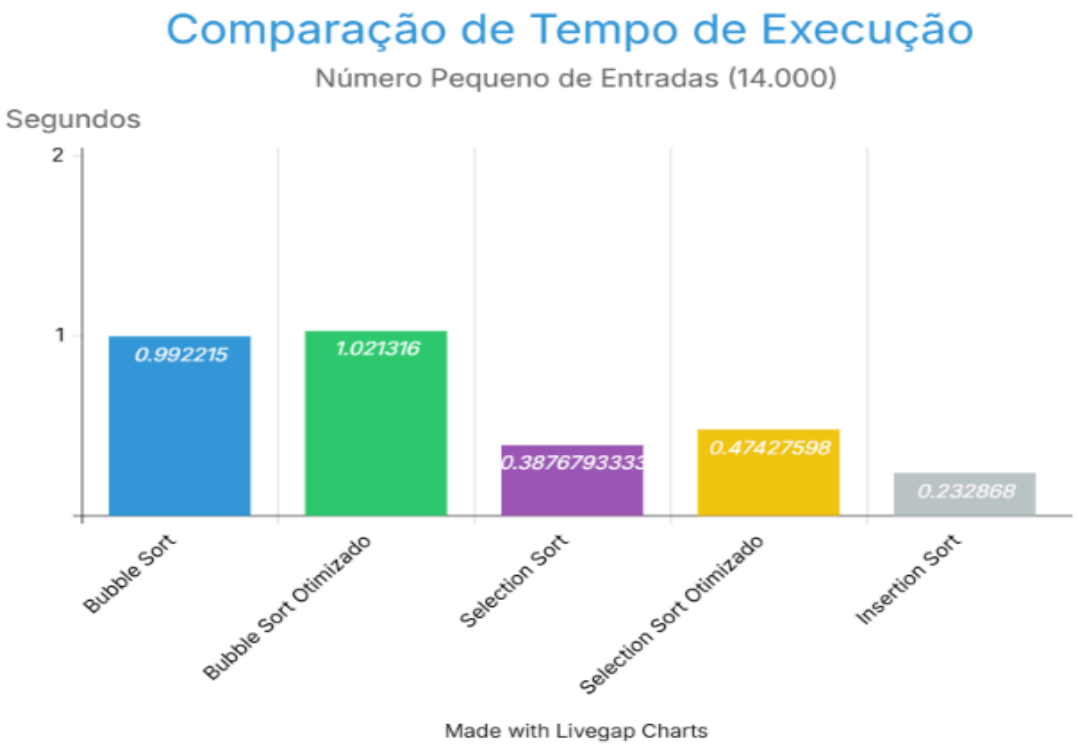
3. Análise dos Resultados

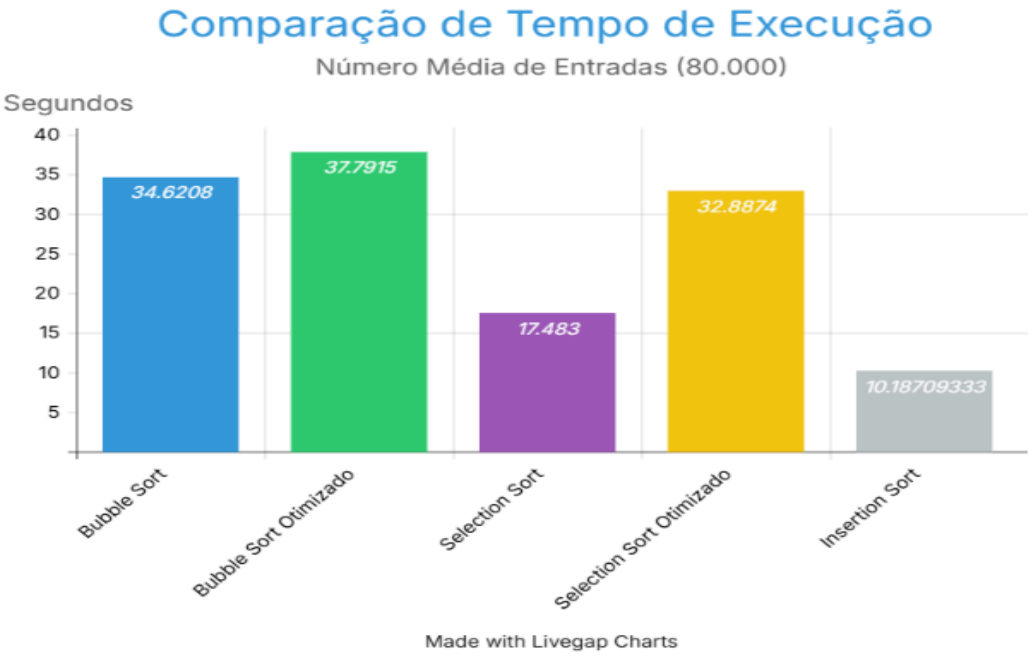
Os resultados foram alcançados analisando a média do tempo de execução de três execuções de seus respectivos códigos.

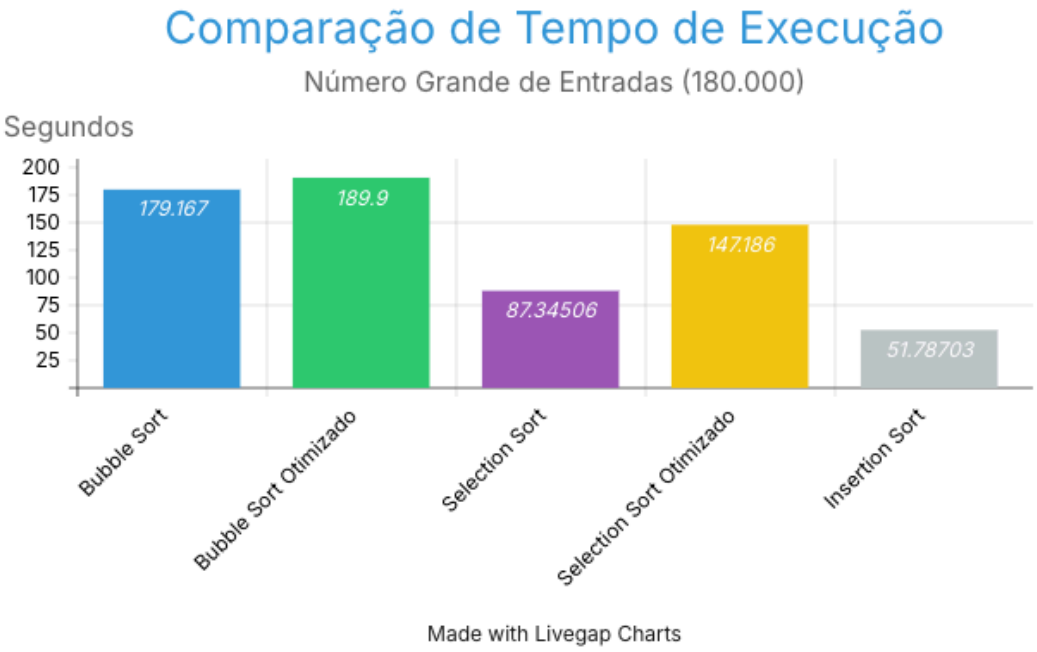
3.1. Algoritmos de Ordenação

3.1.1. Comparação entre os Algoritmos

3.1.1.1. Casos Normais





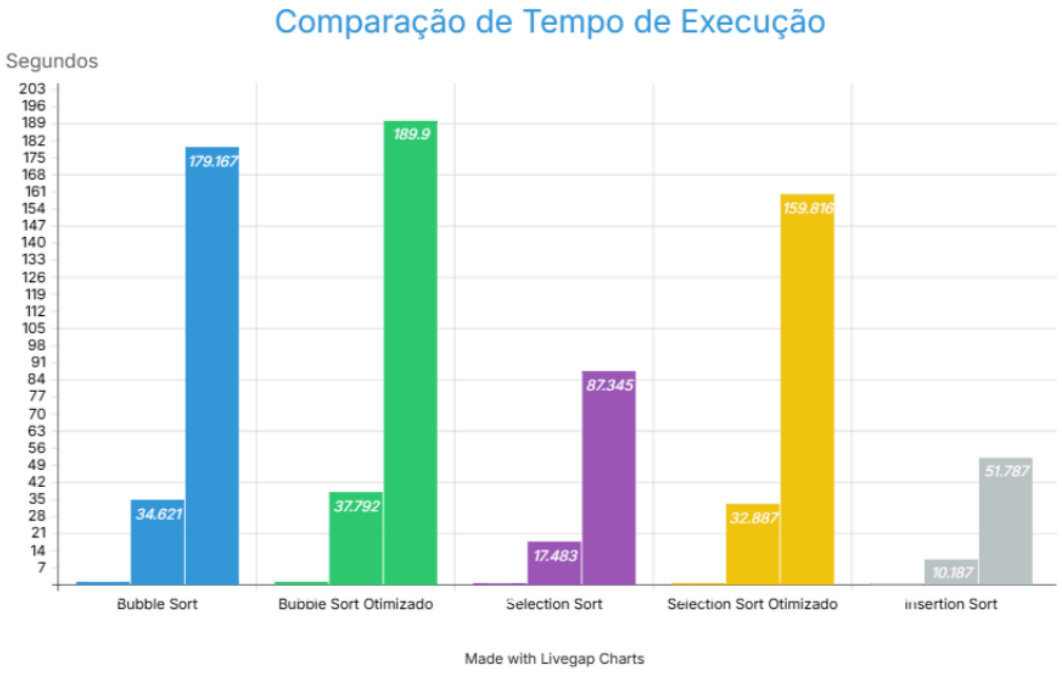


À primeira vista, é fácil dizer quais são os algoritmos mais rápidos. O *Bubble Sort* aparece como consideravelmente mais devagar que todos os outros algoritmos convencionais em todos os testes, se mostrando superior somente à sua versão otimizada, que possui uma comparação a mais por iteração.

O *Selection Sort* se mostrou consideravelmente mais rápido que o *Bubble Sort*. No entanto, sua versão otimizada teve um aumento expressivo do seu tempo de execução, somente se mostrando mais rápido que o *Bubble Sort* e o

Insertion Sort, com os efeitos do seu laço de repetição extra se mostrando mais e mais conforme o número de entradas aumenta.

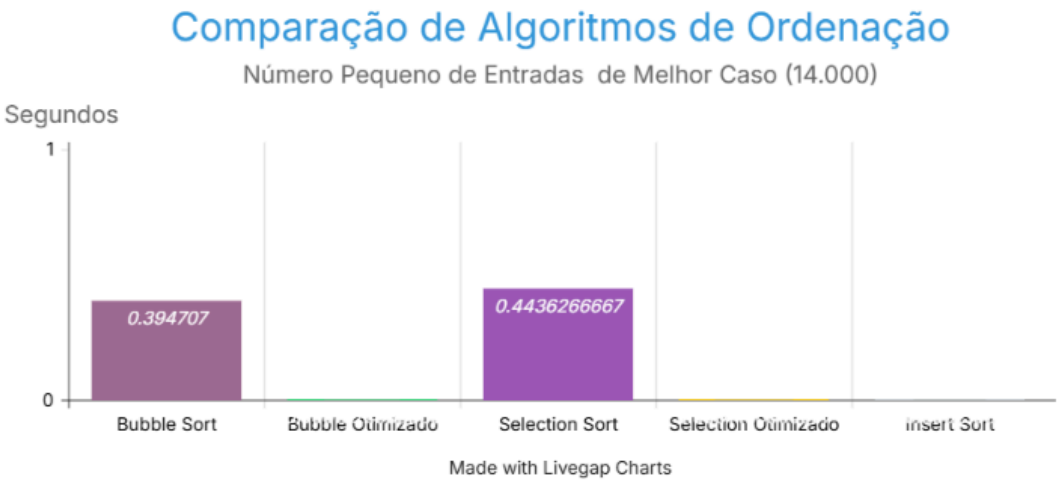
O *Insertion Sort* se mostrou como o mais rápido pela maneira de como realiza sua ordenação, não executando ações desnecessárias e fazendo *shifts* e *inserts* ao invés de *swaps,* que necessitam de 3 atribuições ao invés de uma como nos outros dois.



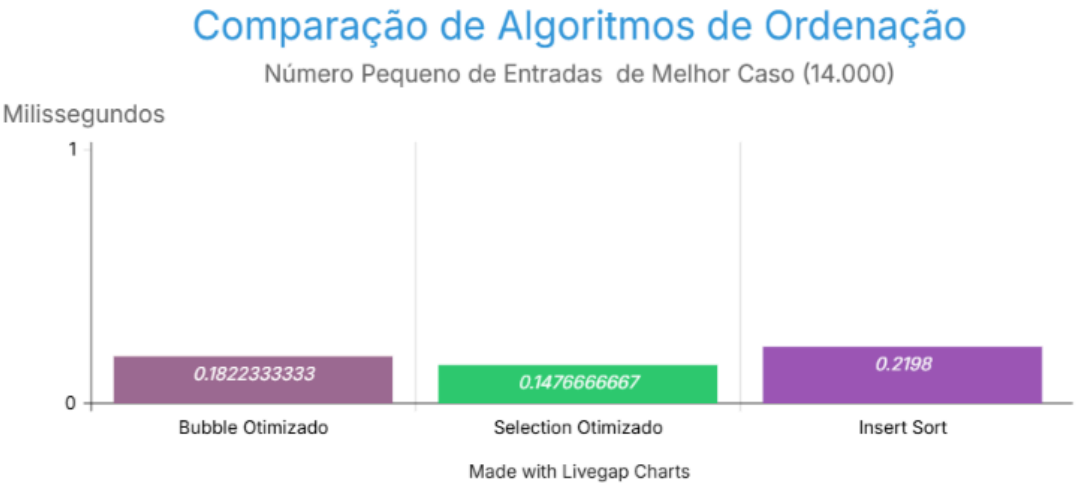
Ao colocar lado a lado as performances em todas as três etapas, além de evidenciar as diferentes velocidades dos algoritmos, como já previamente discutido, a complexidade $O(n^2)$ se torna visível, com os tempos de execução aumentando exponencialmente conforme o número de entradas.

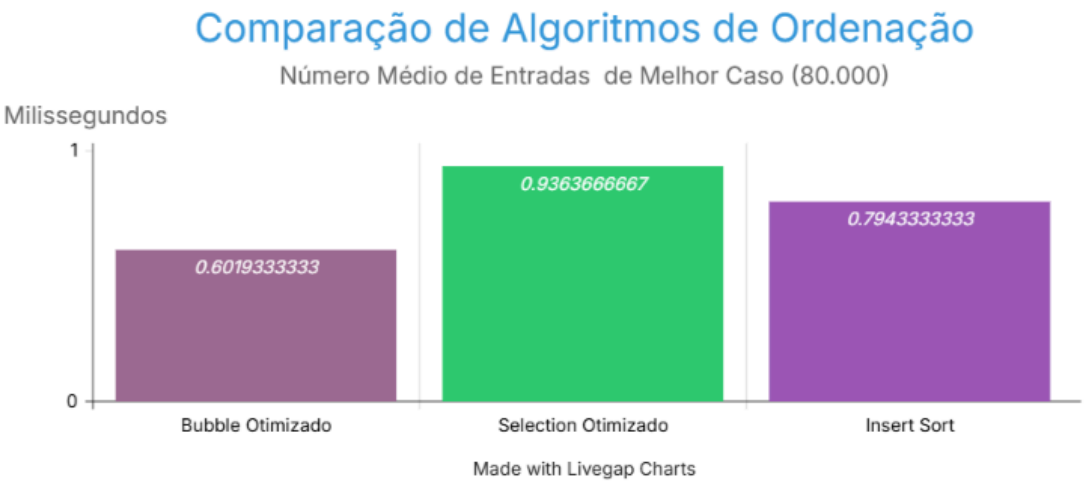
3.1.1.2. Melhores Casos

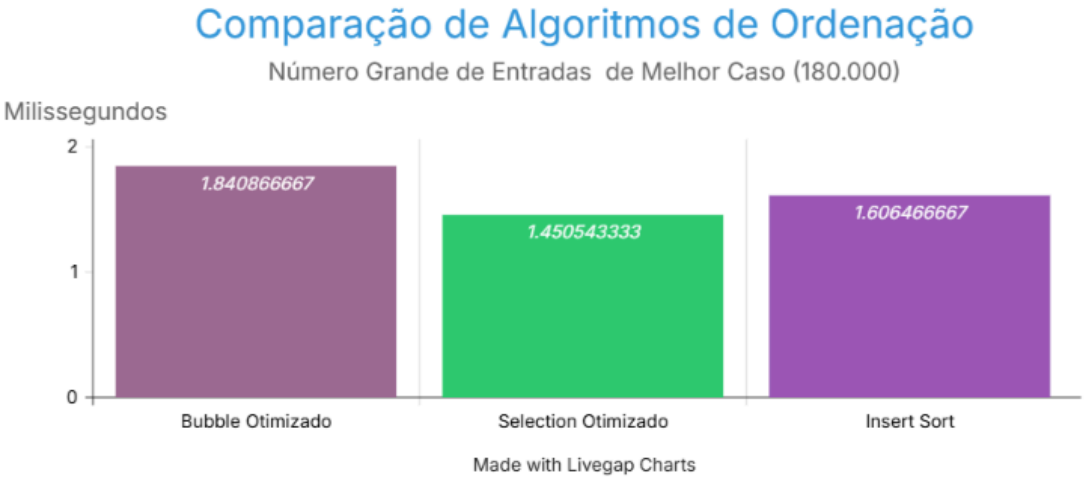
Com os melhores casos, há uma discrepância enorme entre os algoritmos otimizados e o *Insertion Sort* em relação ao *Bubble Sort* e o *Selection Sort* convencionais, já que esses dois não têm a capacidade de sair do laço de repetição principal caso o vetor esteja ordenado ou não entrar no segundo *loop* caso o número da posição atual esteja na ordem correta.



Por causa disso, para que os outros algoritmos se tornem visíveis nos gráficos, a unidade de medida utilizada será *ms* e os resultados do *Bubble Sort* e *Selection Sort* serão desconsiderados, já que eles mantêm por volta do mesmo tempo de execução dos seus casos normais em ambos os grupos de tamanho médio e grande.







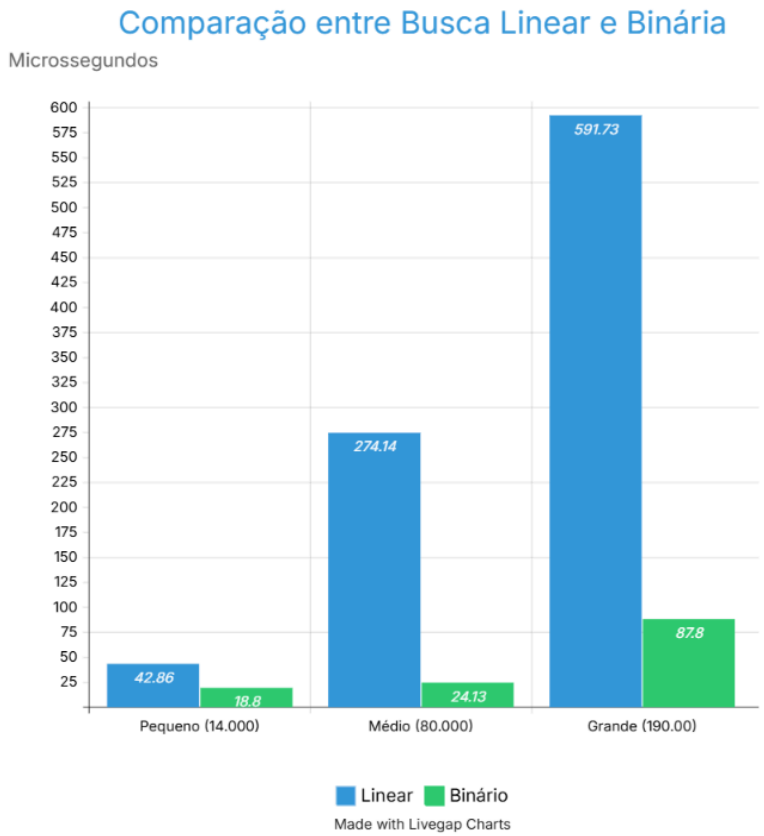
Neles, não há um padrão muito claro, mas no geral, os três algoritmos agem de maneiras muito similares em uma situação de melhor caso, já que todos apresentam entre 0 e 1 trocas e números muito similares de comparações nos melhores casos.

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 que 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.



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 *Insertion Sort* sobre os outros algoritmos de ordenação, isso, como discutido na análise, ocorrendo pelo fato de utilizar *shifts* e *inserts*, que consomem menos recursos, e evitarem laços de repetição desnecessários. A comparação entre os algoritmos de busca também mostrou a diferença entre um algoritmo de complexidade $O(log\ n)$ e $O(n^2)$, apesar da Busca Binária ser utilizável apenas em dados ordenados.

Além disso, as informações obtidas serviram para demonstrar como a complexidade dos algoritmos de fato indicam o seu tempo de execução, com os gráficos mostrando visualmente esse fato.

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. 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.