TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GABRIEL HENRIQUE FURTADO BARBOSA

JOÃO VITOR BEZERRA DA SILVA

RELATÓRIO PARA A DISCIPLINA DE ALGORITMOS

SOBRE ORDENAÇÃO

Natal – RN

2025

INSTITUTO FEDERAL
Rio Grande do Norte
Campus Natal-Central

GABRIEL HENRIQUE FURTADO BARBOSA

JOÃO VITOR BEZERRA DA SILVA

RELATÓRIO PARA A DISCIPLINA DE ALGORITMOS

SOBRE ALGORITMOS DE ORDENAÇÃO

Relatório sobre algoritmos de ordenação para a disciplina de Algoritmos como requisito para nota da segunda etapa do semestre 2024.2.

Discente: Plácido Antônio de Souza Neto.

Natal – RN

2024

Resumo

Esse relatório consiste no projeto de análise de algoritmos de ordenação, são eles, *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort* e *Quick Sort*. Essa análise foi feita em duas linguagens de programação (C e Java) utilizando testes de tempo execução com diferentes quantidades de valores de entrada. O objetivo principal é a busca pelo aperfeiçoamento crítico na analise de algoritmos no que tange as suas eficiências e as suas complexidades. Dessa maneira, foi adquirido o conhecimento das implicações geradas pelas arquiteturas de um algoritmo, a forma mais eficaz de resolução desses empecilhos e a melhor forma de utilizar um algoritmo para cada situação.

Palavras-chaves: Algoritmos, Eficiência, Complexidade, Análise, Ordenação, Implicações.

SUMÁRIO

1. METODOLOGIA

A metodologia resumir-se em implementar nas linguagens de programação C e Java os algoritmos de ordenação descritos. Foi feito testes de tempo de execução os quais foram utilizados para avaliar a eficiência na resolução do problema, ordenar sequências de dados desordenados com 100, 1.000, 10.000, 100.000 e 1.000.000 de valores de entrada. Por meio de quatro (4) testes para cada conjunto de valores foi possível analisar cada algoritmo, gerar gráficos de eficiência e obter o conhecimento de suas implicações em cada caso.

2. ALGORITMOS

Os algoritmos são estruturas de dados que possuem sequências de regras e operações finitas utilizadas para a resolução de problemas. Partindo desse contexto, os algoritmos de ordenações em analise são responsáveis pela resolução do sequenciamento de conjuntos de dados por meio de operações de ordenação. Os algoritmos que estão sendo analisados são: *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort* e *Quick Sort*. Cada algoritmo difere-se um do outro pelos seus métodos para resolução do problema, dessa forma, apesar do mesmo fim, as analises desses algoritmos se dessemelham entre si.

2.1. BUBBLE SORT

O algoritmo *Bubble Sort* é um dos mais simples algoritmos de ordenação. Esse utiliza o método de percorre uma sequência de valores n vezes, sendo n correspondente ao tamanho da sequência, comparando-os um a um e levando a cada passagem o maior elemento para o fim da sequência. Sua complexidade é de ordem quadrática, O (n^2). Em seu melhor caso, sua complexidade é O (n) sendo n operações relevantes executadas, tendo n como os valores da sequência, e em seu pior caso, sua complexidade é O (n^2) sendo n vezes n operações relevantes executadas. Esse algoritmo não possui uma boa eficiência para casos complexos com números elevados de entradas.

2.1.1. Aplicação em C

Figura 1: Código da Aplicação do Algoritmo *Bubble Sort* em C.

```c
int *ordened_array;

void bubble_sort(int length_array)
{
    int i, j, aux;

    for(i = 0; i < length_array; i++)
    {
        for(j = 0; j < length_array-i-1; j++)
        {
            if(ordened_array[j] > ordened_array[j+1])
            {
                aux = ordened_array[j];
                ordened_array[j] = ordened_array[j+1];
                ordened_array[j+1] = aux;
            }
        }
    }
}
```

```c
ordened_array = (int *) malloc(length_array * sizeof(int));
```

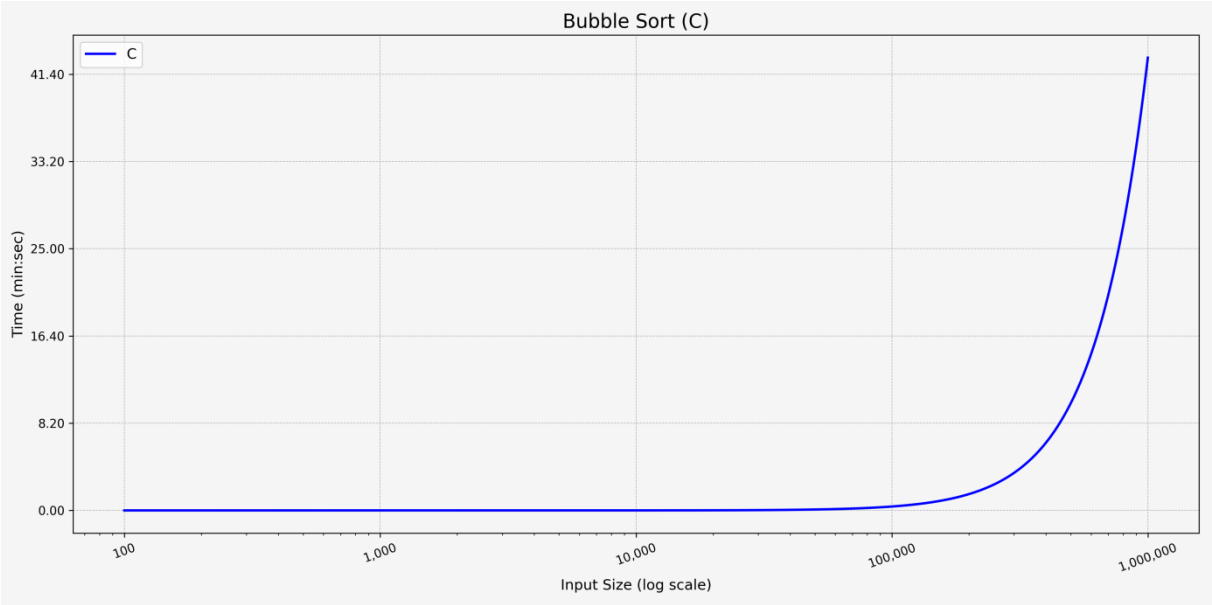
Fonte: código de autoria própria.

Devido a complicações nos testes do algoritmo *Bubble Sort* com grandes entradas, nesse código foi utilizado para os testes de tempo de execução uma versão mais eficiente desse algoritmo, a qual em suas passagens percorre apenas até o valor anterior ao último ordenado. Na linha seis (6), é declarado a variável ponteiro do tipo *int*, *ordened_array*, que é utilizada como um tipo primitivo *array* após ser alocado, na linha quarenta e nove (49), um espaço de memória com o tamanho correspondente ao valor da variável local do tipo *int*, que recebe a quantidade de entradas, *length_array*. Na linha oito (8), a função do tipo *void*, *bubble_sort*, recebe como entrada uma variável do tipo *int*, *length_array*, contendo o tamanho do *array* que irá ser ordenado. Na linha dez (10), são declaradas as variáveis auxiliares *i*, *j* e *aux*. Das linhas doze (12) a vinte e três (23), a primeira instrução de iteração *for* é utilizada

para garantir que todos os elementos do *array* sejam comparados com seus sucessores, percorrendo do valor de inicialização zero (0) ao final do *array*. Das linhas quatorze (14) a vinte e dois (22), a segunda instrução de iteração *for* é utilizada para fazer as comparações de cada elemento do *array* com seus sucessores, como ponto de partida o primeiro valor do array até a penúltima posição em relação a última ordenada, e logo em seguida, na linha dezesseis (16), é feito uma comparação, pela instrução de seleção *if*, sendo verificado se o valor atual é maior que seu sucessor, sendo a comparação verdadeira, é feito a troca de posição entre os dois nas linhas dezoito (18) a vinte (20).

2.1.2. Gráfico de Eficiência em C

Figura 2: Gráfico de Eficiência do algoritmo *Bubble Sort* em C.



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.1.3. Análise em C

Como é mostrado na análise gráfica, o algoritmo *Bubble Sort* possui uma eficiência relativamente boa com valores de entradas baixos, entre cem e dez mil, no entanto, devido a sua construção como algoritmo de ordenação a qual utilizar de n passagens do inicio ao final da sequência de dados para ordena-la, torna-se lento e ineficiente para exercer a sua funcionalidade quando utilizado com grandes quantidades de valores de entrada, entre cem mil e um milhão. Por essa falta de eficiência a analise foi feita com uma versão mais eficiente

desse algoritmo que percorre n vezes a sequência até o penúltimo valor em relação ao último ordenado, porém mesmo com essa alteração o algoritmo *Bubble Sort* demostrou baixa eficácia para cumprir sua função.

2.1.4. Aplicação em Java

Figura 3: Código da Aplicação do Algoritmo *Bubble Sort* em Java.

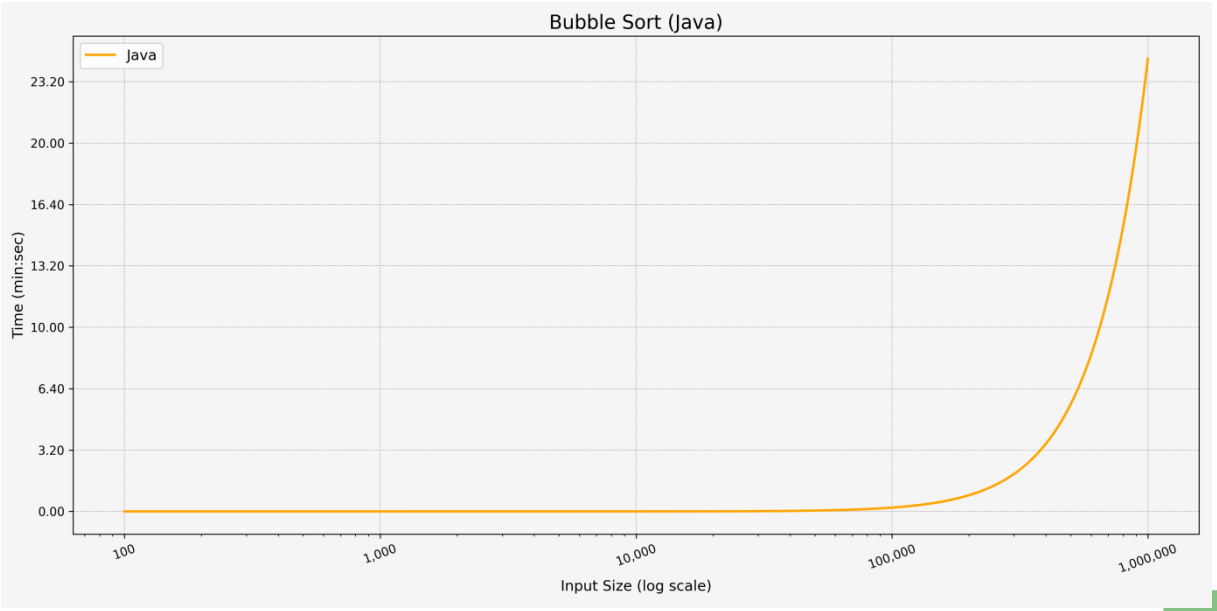
```
 7 class BubbleSort
 8 {
 9     private int[] ordenedList = new int[100];
10     private int lengthList;
11 
12     public BubbleSort(int[] ordenedList, int lengthList)
13     {
14         this.ordenedList = ordenedList;
15         this.lengthList = lengthList;
16     }
17 
18     public int[] OrdenedList()
19     {
20         int i, j, aux;
21 
22         for(i = 0; i < lengthList; i++)
23         {
24             for(j = 0; j < lengthList - i - 1; j++)
25             {
26                 if(ordenedList[j] > ordenedList[j+1])
27                 {
28                     aux = ordenedList[j];
29                     ordenedList[j] = ordenedList[j+1];
30                     ordenedList[j+1] = aux;
31                 }
32             }
33         }
34         return ordenedList;
35     }
36 
```

Fonte: código de autoria própria.

A classe *BubbleSort* é um algoritmo de ordenação, sendo neste caso de números inteiros não exclusivos em ordem crescente, é basicamente composta por dois laços *for* aninhados e tem como parâmetros uma variável do tipo *array* de inteiros (*orderedList*, que foi transformada em atributo construtor na linha 16) e uma variável para definir o tamanho dos elementos desse *array* (*lengthList*, definida como atributo construtor na linha 17). No primeiro laço *for* (localizado na linha 22), a variável do tipo *int* definida como *i* servirá como índice para determinar o índice do primeiro elemento do *array* que será ordenado e terá range de *lengthList*. Já no segundo *for* (localizado na linha 24), a variável do tipo inteiro definida, *j*, servirá para estabelecer uma comparação entre o elemento do *array* com índice *j* e o elemento subsequente, que corresponde ao índice *j+1* e o seu range no laço será de *length -i -1* para evitar que o algoritmo leve mais tempo percorrendo o laço. Caso o elemento de índice *j* seja maior que o elemento de índice *j+1*, os dois atenderão à condição do *if* da linha 28 e terão seus valores trocados com o auxílio da variável *aux*, definida na linha 28 como inteiro. Essa operação ocorrerá "*lengthList* vezes menos um", ao final do processo todo o *array* estará comparado um a um e devidamente ordenado até a conclusão do processo pelo término do primeiro *for*.

2.1.5. Gráfico de Eficiência em Java

Figura 4: Gráfico de Eficiência do algoritmo *Bubble Sort* em Java



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.1.6. Análise em Java

Após ser executado o algoritmo *Bubble Sort* na linguagem Java, percebemos uma baixa variância na faixa de 0.02242 segundos para entradas com potências na região de cem até dez mil. No entanto, para entradas no limite de dez mil a um milhão, essa variância de tempo aumenta abruptamente (chegando a 479302,58 segundos) e tendo um tempo médio de 1474,80 segundos por execução. Dessa forma, pode ser concluído que talvez o *Bubble Sort* não seja o algoritmo de ordenação mais indicado para uma quantidade de entradas elevadas. Porém, mesmo com seus resultados mais demorados em entradas grandes, essa ferramenta se mostrou mais eficiente que o esperado.

2.2. INSERTION SORT

O algoritmo *Insertion Sort* é um simples algoritmos de ordenação. O método utilizado por esse algoritmo é a verificação de um valor pertencente a uma sequência de dados com seus antecessores. Por meio dessa verificação é feita a troca desde que o valor de verificação atual seja menor que o antecessor e seu ponto de parada decorrente do valor atual ser maior que o antecessor. Sua complexidade é de ordem quadrática, O (n^2). Em seu melhor caso, sua complexidade é O (n) sendo n operações relevantes executadas, tendo n como os valores da sequência, e em seu pior caso, sua complexidade é O (n^2) sendo n vezes n operações relevantes executadas. Esse algoritmo não possui uma boa eficiência para casos complexos com números elevados de entradas.

2.2.1. Aplicação em C

Figura 5: Código da Aplicação do Algoritmo *Insertion Sort* em C.

```c
int *ordened_array;

void insertion_sort(int length_array)
{
    int i, temp, aux;

    for(i = 1; i < length_array; i++)
    {
        temp = i;
        while(ordened_array[temp] < ordened_array[temp - 1])
        {
            aux = ordened_array[temp];
            ordened_array[temp] = ordened_array[temp - 1];
            ordened_array[temp - 1] = aux;
            temp--;

            if(temp == 0) break;
        }
    }
}
```

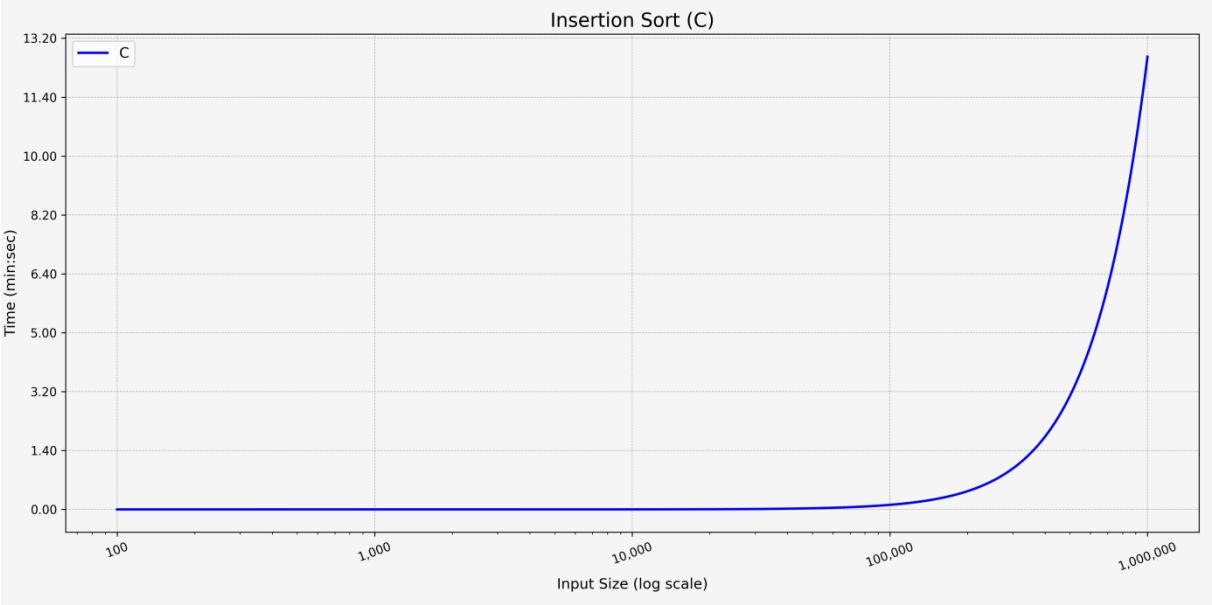
```c
ordened_array = (int *) malloc(length_array * sizeof(int));
```

Fonte: código de autoria própria.

A implementação desse código foi feito baseando-se no método de ordenação de sequências de dados, *Insertion Sort*. Na linha seis (6), é declarado a variável ponteiro do tipo *int*, *ordened_array*, que é utilizada como um tipo primitivo *array* após ser alocado, na linha cinquenta (50), um espaço de memória com o tamanho correspondente ao valor da variável local do tipo *int*, que recebe a quantidade de entradas, *length_array*. Na linha oito (8), a função do tipo *void*, *insertion_sort*, recebe como entrada uma variável do tipo *int*, *length_array*, contendo o tamanho do *array* que irá ser ordenado. Na linha dez (10), são declaradas as variáveis auxiliares *i*, *temp* e *aux*. Das linhas doze (12) a vinte e quatro (24), a primeira instrução de iteração *for* inicializa no valor um (1), correspondente ao mesmo valor para o índice do *array* a ser ordenado, e percorre todos os valores seguintes. Na linha quatorze (14), a variável temporária *temp* recebe o valor da posição atual de referência do *array*. Das linhas quinze (15) a vinte e três (23), a segunda instrução de iteração *while* é utilizada para fazer as comparações do valor de referência com seus antecessores. Sua condição de parada é quando o valor de verificação atual torna-se a ser maior que seu antecessor. Enquanto essa condição não perecer, será efetuada a troca do valor de referência com seu antecessor nas linhas dezessete (17) a dezenove (19). A linha vinte (20) faz o decremento do valor da variável temporária que passa, logo em seguida, pela outra verificação de parada, estrutura de seleção *if*, na linha vinte e dois (22) que consiste na saída da estrutura de iteração quando o valor da variável temporária chega a zero (0).

2.2.2. Gráfico de Eficiência em C

Figura 6: Gráfico de Eficiência do algoritmo *Inserion Sort* em C.



Fonte: gráfico gerado por algoritmo em linguagem Python.

2.2.3. Análise em C

Com base na análise gráfica, o algoritmo *Insertion Sort* apresentou uma melhor de aproximadamente 69% em relação ao algoritmo anterior, *Bubble Sort*, ocorrendo uma redução de aproximadamente 28 minutos em relação ao tempo gasto para a ordenação da sequência de um milhão de entradas. Apesar da melhor eficiência o algoritmo ainda apresentou ineficiência para entradas entre cem mil e um milhão de dados, isso é mostrado em sua complexidade O (n^2). Isso pode ser explicado devido a suas n passagens na sequência comparando-a da posição de referência com seus antecessores até o seu inicio.

2.2.4. Aplicação em Java

Figura 7: Código da Aplicação do Algoritmo *Insertion Sort* em Java.

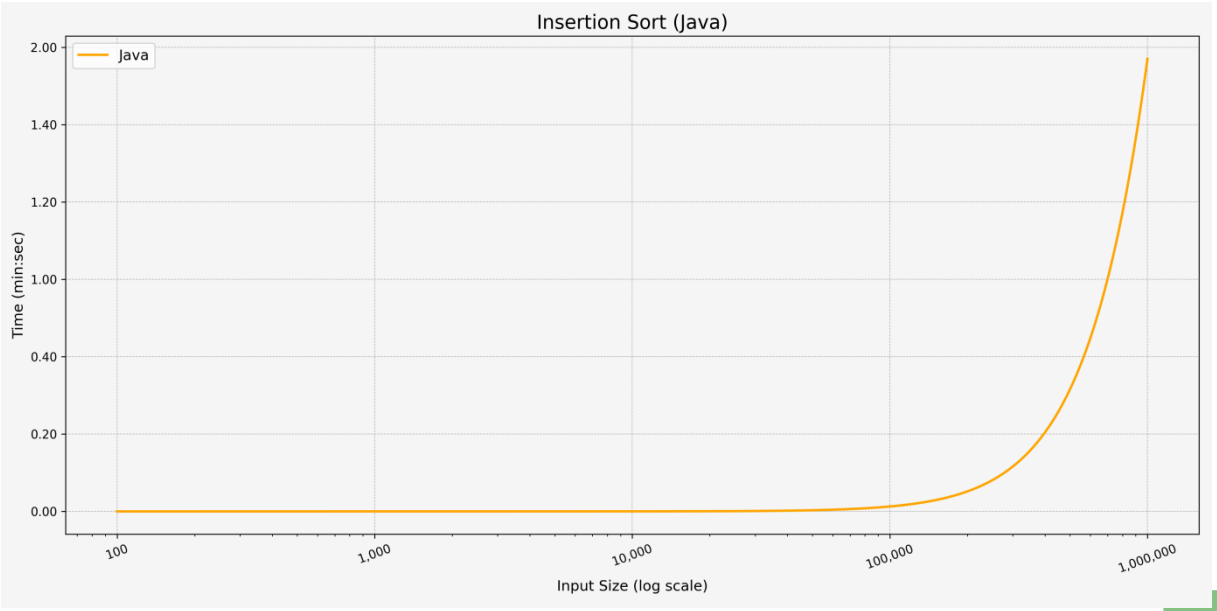
```java
class InsertionSort
{
    private int[] ordenedList = new int[100];
    private int lengthList;

    public InsertionSort(int[] ordenedList, int lengthList)
    {
        this.ordenedList = ordenedList;
        this.lengthList = lengthList;
    }

    public int[] OrdenedList()
    {
        int i, j, aux;

        for(i = 1; i < lengthList; i++)
        {
            j = i;
            while (j > 0 && ordenedList[j] < ordenedList[j-1])
            {
                aux = ordenedList[j];
                ordenedList[j] = ordenedList[j-1];
                ordenedList[j-1] = aux;
                j--;
            }
        }
        return ordenedList;
    }

```

Fonte: código de autoria própria.

A classe *InsertionSort*, que implementa o algoritmo de ordenação *Insertion Sort*, tem como objetivo, da mesma forma que o *Bubblesort* em Java, organizar números inteiros não exclusivos em ordem crescente. Ela é composta principalmente por um laço *for* e um laço *while* aninhado, tendo como parâmetros um *array* de inteiros (*ordenedList*, transformado em atributo construtor na linha 8) e um valor inteiro para definir o tamanho desse *array* (*lengthList*, definido como atributo construtor na linha 9). O laço *for* principal (linha 14) começa a partir do segundo elemento do *array* (índice um), pois o primeiro elemento já é considerado ordenado. A variável inteira *i* é usada para percorrer o *array*, indicando o elemento que será inserido corretamente na parte já ordenada da lista. Dentro do *for*, a variável inteira *j* (linha 16) é utilizada para comparar o elemento atual (*ordenedList[j]*) com os elementos anteriores da lista. O laço *while* (linha 18) verifica se o valor atual é menor que o valor anterior (*ordenedList[j-1]*) e, caso positivo, realiza a troca de valores utilizando a variável *aux* (linha 19), deslocando os elementos maiores para frente e posicionando corretamente o menor valor. Esse processo se repete até que todos os elementos estejam ordenados. O resultado final é uma lista ordenada, retornada pela variável *ordenedList* na linha 25.

2.2.5. Gráfico de Eficiência em Java

Figura 8: Gráfico de Eficiência do algoritmo *Inserion Sort* em Java.



Fonte: gráfico gerado por algoritmo em linguagem Python.

2.2.6. Análise em Java

Após executar o algoritmo *Insertion Sort* na linguagem Java, foi observada uma grande melhora de eficiência se comparada ao algoritmo anterior (o *Bubble Sort*, que também apresenta uma complexidade O (n^2)). Isso pode ser visto em entradas com valores entre cem a cem mil, apresentando uma variância de apenas 0,2392 segundos. Ademais, com entradas no valor de um milhão, o algoritmo demorou em média 109,18 segundos, um valor satisfatório de tempo e irrelevante, se comparado ao código antecessor.

2.3. SELECTION SORT

O algoritmo *Selection Sort* é um dos algoritmos de ordenação mais simples com uma eficiência relativamente boa. Esse algoritmo utiliza o método de percorre uma sequência de valores em busca do menor termo da lista e levando a cada passagem esse menor valor encontrado para o índice em análise naquele momento. Sua complexidade é de ordem quadrática O (n^2) em todos os casos (no seu melhor e no seu pior caso), pois independentemente do número de entradas o código precisará realizar “[n(n-1)]/2” operações, sendo n a quantidade de valores da sequência. Esse algoritmo não possui uma boa estabilidade com grandes entradas de dados.

2.3.1. Aplicação em C

Figura 9: Código da Aplicação do Algoritmo *Selection Sort* em C.

```c
int *ordened_array;

void selection_sort(int length_array)
{
    int i, j, temp, aux;

    for(i = 0; i < length_array - 1; i++)
    {
        temp = i;
        for(j = i + 1; j < length_array; j++)
        {
            if(ordened_array[j] < ordened_array[temp]) temp = j;
        }
        if(temp != i)
        {
            aux = ordened_array[i];
            ordened_array[i] = ordened_array[temp];
            ordened_array[temp] = aux;
        }
    }
}
```

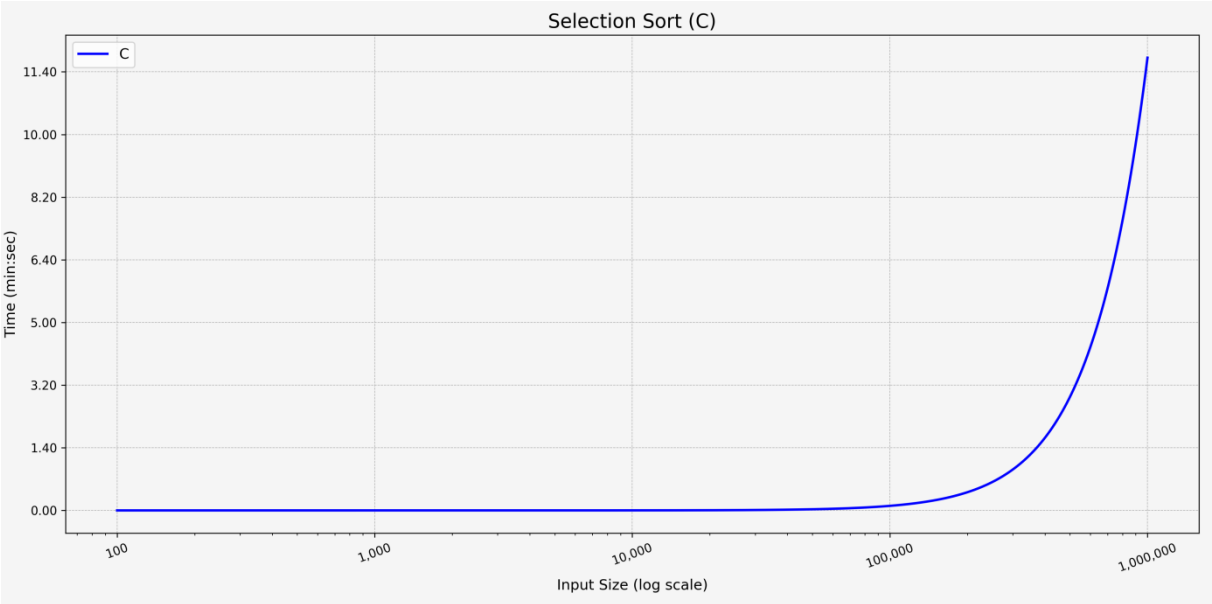
```c
ordened_array = (int *) malloc(length_array * sizeof(int));
```

Fonte: código de autoria própria.

A implementação desse código foi feito baseando-se no método de ordenação de sequências de dados, *Selection Sort*. Na linha seis (6), é declarado a variável ponteiro do tipo *int*, *ordened_array*, que é utilizada como um tipo primitivo *array* após ser alocado, na linha cinquenta (51), um espaço de memória com o tamanho correspondente ao valor da variável local do tipo *int*, que recebe a quantidade de entradas, *length_array*. Na linha oito (8), a função do tipo *void*, *selection_sort*, recebe como entrada uma variável do tipo *int*, *length_array*, contendo o tamanho do *array* que irá ser ordenado. Na linha dez (10), são declaradas as variáveis auxiliares *i*, *j*, *temp* e *aux*. Das linhas doze (12) a vinte e quatro (25), a primeira instrução de iteração *for* inicializa no valor zero (0), correspondente ao mesmo valor para o índice do *array* a ser ordenado, e percorre todos os valores seguintes até o penúltimo valor do *array*. Na linha quatorze (14), a variável temporária *temp* recebe o valor da posição atual de referência do *array*, *i*. Das linhas quinze (15) a vinte e três (23), a segunda instrução de iteração *for* é responsável por busca à posição correta do valor de referência partido do seu sucessor até o final do *array*. Na linha dezessete (17), ocorre a troca do valor de posição do valor de referência quando o seu sucessor for menor e nas linhas dezenove (19) a vinte quatro (24) ocorre à troca, se o valor de referência não estiver ordenado, da sua posição com o valor que está em sua verdadeira posição.

2.3.2. Gráfico de Eficiência em C

Figura 10: Gráfico de Eficiência do algoritmo *Selection Sort* em C.



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.3.3. Análise em C

Com base na análise gráfica percebe-se que o algoritmo *Selection Sort* possui uma eficiência, aproximadamente, 74% melhor que o *Bubble Sort* e, aproximadamente, 16% melhor que o *Insertion Sort*. Essa melhora de eficiência ocorreu devido a sua arquitetura como algoritmo de ordenação que basear-se em fazer n passagens pela sequência de dados sempre eliminando o ultimo valor ordenado, dessa forma torna-o um pouco mais eficiente que o algoritmo anterior, *Insertion Sort*. No entanto, assim como os algoritmos anteriores, o *Selection Sort* perde sua eficiência quando usado com grandes entradas de dados, entre cem mil e um milhão, dessa forma é correto afirma que esse algoritmo não é recomendado para a ordenação de volumosos números de dados.

2.3.4. Aplicação em Java

Figura 11: Código da Aplicação do Algoritmo *Selection Sort* em Java.

```java
class SelectionSort {
    private int[] ordenedList = new int[100];
    private int lengthList;

    public SelectionSort(int[] ordenedList, int lengthList)
    {
        this.ordenedList = ordenedList;
        this.lengthList = lengthList;
    }

    public int[] OrdenedList()
    {
        int i, j, min, aux;

        for(i = 0; i < (lengthList-1); i++)
        {
            min = i;
            for(j = (i+1); j < lengthList; j++)
            {
                if(ordenedList[j] < ordenedList[min]) min = j;
            }
            if (i != min)
            {
                aux = ordenedList[i];
                ordenedList[i] = ordenedList[min];
                ordenedList[min] = aux;
            }
        }
        return ordenedList;
    }

```

Fonte: código de autoria própria.

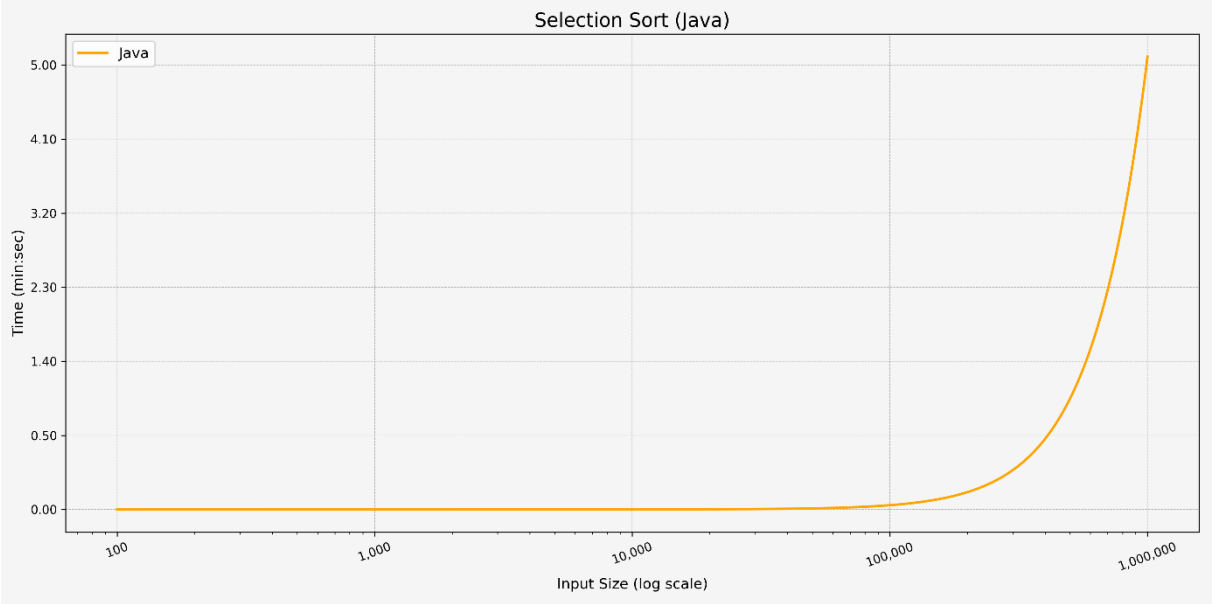
A implementação desse código foi feita com base no método de ordenação de sequências de dados *Selection Sort*. Nas linhas 8 e 9, são declaradas as variáveis: um *array* de inteiros (*ordenedList*, representando a lista que será ordenada ao final da execução) e uma variável do tipo inteiro (*lengthList*), representando o tamanho da lista que será ordenada.

Nas linhas 17 a 36, foi criado um método chamado *OrdenedList*, que retorna uma lista de inteiros. Na linha 19, foram declaradas quatro variáveis (*i*, *j*, *min* e *aux*) com o objetivo de contribuir para a execução da ordenação dos números. Mas de que forma? Logo após a declaração dessas variáveis, é realizado um laço for que percorre de *i* igual zero até *lengthList - 1*. A variável *i* é atribuída à variável min na linha 23 para sinalizar o índice do próximo valor mínimo a ser ordenado.

Dessa forma, ao executar o segundo laço for, em vez de iniciar *j* igual zero, ele recebe o valor *j* igual *i + 1*, garantindo que sempre seja analisado o valor do *array* nos índices superiores ao de *i*, que deve conter o menor valor. Dentro desse laço, há uma condição que verifica se *ordenedList[j]* é menor que *ordenedList[min]*. Se for, min recebe o índice *j*. Além disso, há outra condicional nas linhas 28 a 33 para verificar se o valor do índice registrado em *i* é diferente do valor do índice registrado em min. Caso sejam diferentes, ocorre a troca entre o valor do *array* na posição *i* e o valor do *array* na posição min, que recebeu o valor de *j*. Essa operação é executada *n(n-1)/2* vezes, que corresponde à soma aritmética de todas as comparações realizadas ao longo do programa. Por fim, na linha 35, o método retorna a lista com os valores ordenados em ordem crescente.

2.3.5. Gráfico de Eficiência em Java

Figura 12: Gráfico de Eficiência do algoritmo *Selection Sort* em Java.



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.3.6. Análise em Java

Como já foi afirmado anteriormente, o *Selection Sort* é um algoritmo de ordenação eficiente. Porém talvez não seja a melhor escolha para grandes grupos de dados, visto que independente do número de entradas a sua eficiência será de ordem quadrática, O (n^2). Entretanto, se compararmos o mesmo algoritmo nas linguagens de programação C e Java, o *Selection Sort* se mostrou mais eficiente na linguagem Java para entradas acima de 100 mil valores. Provavelmente isso foi ocasionado pela otimização natural da linguagem de programação Java em relação à linguagem de programação C (sem utilizar otimizadores).

2.4. MERGE SORT

O algoritmo *Merge Sort* é um dos algoritmos de ordenação mais eficientes, baseado no paradigma de divisão e conquista. Esse método divide a sequência de valores em pequenas partes, ordena cada uma delas separadamente e, em seguida, combina-as de forma ordenada. Sua complexidade é de ordem O (n log n) em todos os casos (melhor, pior e médio caso), tornando-o mais eficiente do que algoritmos de ordem quadrática para grandes conjuntos de dados. Esse algoritmo possui boa estabilidade, pois mantém a ordem relativa dos elementos iguais. Além disso, seu desempenho é consistente, pois, mesmo com um número grande de entradas, ele realiza um número previsível de operações devido à sua abordagem recursiva e à fusão ordenada das sublistas.

2.4.1. Aplicação em C

Figura 13: Código da Aplicação do Algoritmo *Merge Sort* em C.

```c
int *ordened_array;
int *vetAux;

void merge(int begin, int middle, int end)
{
    int auxBegin = begin, auxMiddle = middle+1, aux = 0, size = end-begin+1;
    vetAux = (int*)malloc(size * sizeof(int));

    while(auxBegin <= middle && auxMiddle <= end)
    {
        if(ordened_array[auxBegin] < ordened_array[auxMiddle])
        {
            vetAux[aux] = ordened_array[auxBegin];
            auxBegin++;
        } else {
            vetAux[aux] = ordened_array[auxMiddle];
            auxMiddle++;
        }
        aux++;
    }

    while(auxBegin <= middle)
    {
        vetAux[aux] = ordened_array[auxBegin];
        auxBegin++;
        aux++;
    }

    while(auxMiddle <= end)
    {
        vetAux[aux] = ordened_array[auxMiddle];
        auxMiddle++;
        aux++;
    }

    for(aux = begin; aux <= end; aux++)
    {
        ordened_array[aux] = vetAux[aux-begin];
    }

    free(vetAux);
}

void merge_sort(int begin, int end)
{
    if (begin < end) {
        int middle = (end+begin)/2;

        merge_sort(begin, middle);
        merge_sort(middle+1, end);
        merge(begin, middle, end);
    }
}
```

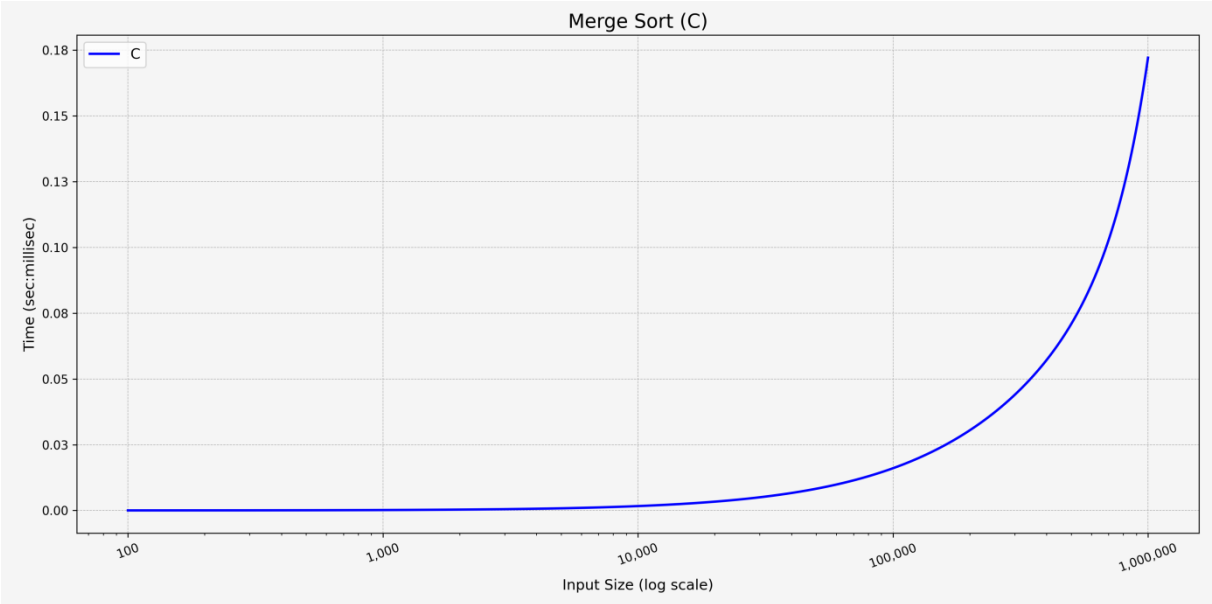
```
83  ordened_array = (int *) malloc(length_array * sizeof(int));
```

Fonte: código de autoria própria.

A implementação desse código foi feito baseando-se no método de ordenação de sequências de dados, *Merge Sort*. Na linha seis (6), é declarado a variável ponteiro do tipo *int*, *ordened_array*, que é utilizada como um tipo primitivo *array* após ser alocado, na linha oitenta e três (83), um espaço de memória com o tamanho correspondente ao valor da variável local do tipo *int*, que recebe a quantidade de entradas, *length_array*. Na linha sete (7), é declarado a variável ponteiro do tipo *int*, *vetAux*, que será utilizado como tipo primitivo *array* auxiliar, após alocamento de memoria feito na função *merge* na linha doze (12). A função *merge_sort*, linha quarenta e nove (49), recebe as entradas *begin*, correspondente a referência de inicio do *array*, e *end*, correspondente a referência do final do *array*. Nas linhas cinquenta e um (51) a cinquenta e sete (57) é feito as chamadas recursivas utilizadas para dividir ao meio o *array ordened_array* em *subarrays* até esses possuírem um elemento. Na linha cinquenta e dois (52) é declarada a variável do tipo *int*, *middle*, correspondente a posição do meio do *array*. Nas linhas cinquenta e quatro (54) e cinquenta e cinco (55) possui a divisão do *array* no meio e na linha cinquenta e seis (56) é chamada a função *merge* que possui com valores de entrada *begin*, correspondente a posição de inicio do *array*, *middle*, correspondente a posição do meio do *array*, e *end*, correspondente a posição final do *array*. Na linha onze (11) são declaradas as variáveis do tipo *int auxBegin*, correspondente ao auxiliar da posição de inicio, *auxMiddle*, correspondente a posição auxiliar do inicio depois da metade, *aux*, correspondente a variável auxiliar, e *size*, correspondente ao tamanho do *array* auxiliar, *vetAux*. Nas linhas quatorze (14) a vinte e cinco (25) é feito por meio da estrutura de iteração *while* a adição ao *array* auxiliar do menor valor entre a primeira e segunda metade do *array* até uma das partes chegarem ao fim. Nas linhas vinte e sete (27) a trinta e nove (39) as estruturas de iteração *while* são responsáveis por fazer a adição do resto, se houve, dos valores de uma das metades. Por fim, nas linhas quarenta e um (41) a quarenta e quatro (44) a estrutura de iteração *for* e responsável por passar os valores do *array* auxiliar para o *array* principal e na linha quarenta e seis (46) é liberado o espaço de memoria correspondente ao *array* auxiliar.

2.4.2. Gráfico de Eficiência em C

Figura 14: Gráfico de Eficiência do algoritmo *Merge Sort* em C.



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.4.3. Análise em C

Com base na análise gráfica, o algoritmo *Merge Sort* possui uma eficiência, aproximadamente, 99,99% maior que o *Bubble Sort* e, aproximadamente, 99,98% maior que o *Insertion Sort* e o *Selection Sort*. Essa melhora significativa na eficiência desse algoritmo em comparação com os anteriores está no modo como são usadas as chamadas recursivas, as quais dividem a ordenação da sequência de dados pela metade e consequentemente reduzindo grandemente o tempo de execução da ordenação. Dessa forma, em divergência com os outros algoritmos analisados, o *Merge Sort* possui uma excelente eficiência para grandes quantidades de valores de entrada, entre cem mil e um milhão.

2.4.4. Aplicação em Java

Figura 15: Código da Aplicação do Algoritmo *Merge Sort* em Java.

```java
class MergeSort {
    public int[] ordenedList = new int[100];

    public MergeSort(int[] ordenedList) {
        this.ordenedList = ordenedList;
    }

    public void OrdenedList(int begin, int end)
    {
        if (begin < end) {
            int middle = (end+begin)/2;

            OrdenedList(begin, middle);
            OrdenedList(middle+1, end);
            Merge(begin, middle, end);
        }
    }

    public void Merge(int begin, int middle, int end)
    {
        int auxBegin = begin, auxMiddle = middle+1, aux = 0, size = end-begin+1;
        int[] vetAux = new int[size];

        while(auxBegin <= middle && auxMiddle <= end)
        {
            if(ordenedList[auxBegin] < ordenedList[auxMiddle])
            {
                vetAux[aux] = ordenedList[auxBegin];
                auxBegin++;
            } else {
                vetAux[aux] = ordenedList[auxMiddle];
                auxMiddle++;
            }
            aux++;
        }

        while(auxBegin <= middle)
        {
            vetAux[aux] = ordenedList[auxBegin];
            auxBegin++;
            aux++;
        }

        while(auxMiddle <= end)
        {
            vetAux[aux] = ordenedList[auxMiddle];
            auxMiddle++;
            aux++;
        }

        for(aux = begin; aux <= end; aux++)
        {
            ordenedList[aux] = vetAux[aux-begin];
        }
    }

```

Fonte: código de autoria própria.

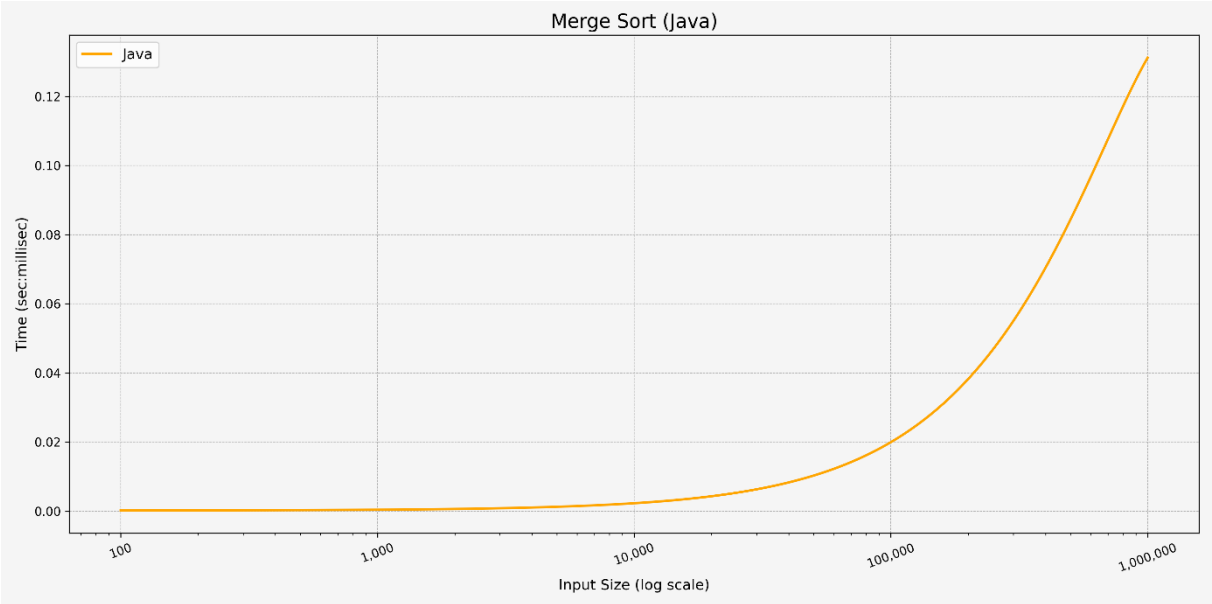
Nas linhas 8 a 11, a variável *ordenedList* é declarada como um *array* de inteiros, que representa a lista de números a ser ordenada. O construtor *MergeSort(int[] ordenedList)* recebe esse *array* e o inicializa.

O método *OrdenedList(int begin, int end)***,** na linha 25, é responsável pela divisão recursiva da lista. Ele recebe dois índices (*begin* e *end*) que representam os limites da lista a ser ordenada. Se o índice inicial for menor que o índice final, o método calcula o índice do meio da lista e chama recursivamente a si mesmo para dividir a lista em duas partes: da posição *begin* à *middle*, e de *middle + 1* até *end*. Após a divisão das sublistas, o método *Merge(int begin, int middle, int end)* é chamado para realizar a fusão dessas partes, ordenando-as ao longo do processo.

No método *Merge*, duas sublistas são combinadas em uma lista ordenada. O método usa três índices auxiliares: *auxBegin*, *auxMiddle* e *aux*. O processo de fusão é realizado comparando os elementos das duas sublistas, movendo o menor valor para o *array* auxiliar *vetAux*. Caso todos os elementos de uma sublista sejam inseridos no *array* auxiliar antes da outra, os elementos restantes da segunda sublista são copiados diretamente para *vetAux*. Após a fusão, os elementos ordenados de *vetAux* são copiados de volta para a lista original *ordenedList*.

2.4.5. Gráfico de Eficiência em Java

Figura 16: Gráfico de Eficiência do algoritmo *Merge Sort* em Java.



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.4.6. Análise em Java

Com base no que foi observado pode-se concluir que, por se utilizar o método de "dividir para conquistar" no *Merge Sort* unida a recursividade, a sua eficiência aumentou exponencialmente se comparado à eficiência dos algoritmos anteriores, *Bubble*, *Insertion* e *Selection Sort*. Ademais, fazendo uma análise numérica entre o mesmo algoritmo nas linguagens C e Java, é possível concluir que a linguagem C se mostrou mais eficiente para valores de entrada até cem mil. Porém, com quantidades de entrada iguais ou superiores a um milhão a linguagem Java se mostrou mais ágil.

2.5. QUICK SORT

O algoritmo *Quick Sort* é um dos algoritmos de ordenação mais rápidos e mais eficientes, porém possui um método de ordenação não estável. Assim como o *Merge Sort*, esse algoritmo possui a estratégia de "dividir para conquistar". É declarado um elemento de pivô o qual é utilizado como referencia para fazer a partição da sequência de dados em duas partes: a primeira contendo os elementos menores que o pivô e a segunda os maiores. E por

fim, por meio de uma chamada recursiva ordenar as duas partes. Esse algoritmo possui como complexidades O (n^2) no pior caso e O (n log n) no médio e melhor caso.

2.5.1. Aplicação em C

Figura 17: Código da Aplicação do Algoritmo *Quick Sort* em C.

```c
int *ordened_array;

void quick_sort(int begin, int end)
{
    int i, j, pivo, aux;
    i = begin;
    j = end-1;
    pivo = ordened_array[(begin + end) / 2];

    while(i <= j)
    {
        while(ordened_array[i] < pivo && i < end)
        {
            i++;
        }
        while(ordened_array[j] > pivo && j > begin)
        {
            j--;
        }

        if(i <= j)
        {
            aux = ordened_array[i];
            ordened_array[i] = ordened_array[j];
            ordened_array[j] = aux;
            i++;
            j--;
        }
    }

    if(j > begin) quick_sort(begin, j+1);
    if(i < end) quick_sort(i, end);
}
```

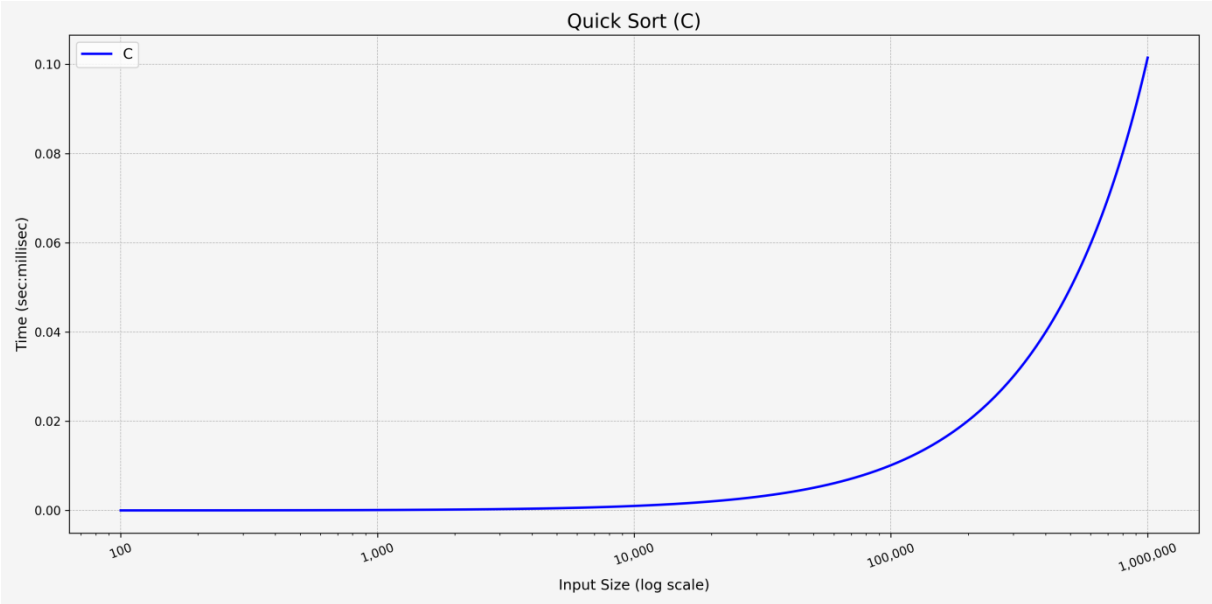
```
63 ordened_array = (int *) malloc(length_array * sizeof(int));
```

Fonte: código de autoria própria.

A implementação desse código foi feito baseando-se no método de ordenação de sequências de dados, *Quick Sort*. Na linha seis (6), é declarado a variável ponteiro do tipo *int*, *ordened_array*, que é utilizada como um tipo primitivo *array* após ser alocado, na linha sessenta e três (63), um espaço de memória com o tamanho correspondente ao valor da variável local do tipo *int*, que recebe a quantidade de entradas, *length_array*. Na linha oito (8), a função *quick_sort* recebe dois valores de entrada: *begin*, correspondente a referência da posição de inicio do *array*, e *end*, correspondente a referência da posição final do *array*. Na linha dez (10), são declaradas as variáveis do tipo *int i* com a posição inicial *begin*, *j* com a posição final *end* menos um (1), *pivo* correspondente ao valor da posição *begin* menos *end* divido por dois (2) e *aux* como variável auxiliar. A primeira instrução de iteração *while*, linhas quinze (15) a trinta e quatro (34) é responsável por separar os valores menores que o pivô à esquerda e os maiores a direita. Nas instruções de iteração *while* mais internas têm a funcionalidade de obter, na primeira iteração com referencia no inicio do *array*, a posição do valor maior que o pivô e, na segunda iteração com referencia no final do *array*, a posição do valor menor que o pivô, tendo ambas como ponto de parada o final e o inicio do *array*, respectivamente. Nas linhas vinte e três (23) a trinta e três (33) ocorre, se a variável de referencia da posição de inicio *i* for menor ou igual a variável de referencia da posição final *j*, a troca dos valores das posições encontradas anteriormente. Nas linhas trinta e seis (36) e trinta e sete (37) acontece as duas chamadas recursivas para o caso do valor de referência final *j* for maior que inicio, *begin*, ou se o valor de referência de inicio *i* for maior que o final, *end*.

2.5.2. Gráfico de Eficiência em C

Figura 18: Gráfico de Eficiência do algoritmo *Quick Sort* em C.



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.5.3. Análise em C

Com base na análise gráfica podemos inferir que o algoritmo *Quick Sort* possui uma eficiência, aproximadamente, 99,99% melhor que o *Bubble Sort*, *Insertion Sort* e *Selection Sort* e, aproximadamente, 45% melhor que o *Merge Sort* para ordenação de grandes quantidades de dados. Essa diferença de eficiência em relação aos outros algoritmos torna-o o melhor algoritmo de ordenação dentre os analisados, no tocante a eficiência em tempo de execução. A explicação dessa análise esta relacionada ao modo como o algoritmo utiliza do pivô para conseguir dividir a sequencia de dados em duas partes contendo os menores e maiores valores, em relação ao pivô, separados.

2.5.4. Aplicação em Java

Figura 19: Código da Aplicação do Algoritmo *Quick Sort* em Java.

```
class QuickSort {
    public int[] ordenedList = new int[100];

    public QuickSort(int[] ordenedList) {
        this.ordenedList = ordenedList;
    }

    public void OrdenedList(int begin, int end)
    {
        int i, j, pivo, aux;
        i = begin;
        j = end-1;
        pivo = ordenedList[(begin + end) / 2];

        while(i <= j)
        {
            while(ordenedList[i] < pivo && i < end)
            {
                i++;
            }
            while(ordenedList[j] > pivo && j > begin)
            {
                j--;
            }

            if(i <= j)
            {
                aux = ordenedList[i];
                ordenedList[i] = ordenedList[j];
                ordenedList[j] = aux;
                i++;
                j--;
            }
        }

        if(j > begin) OrdenedList(begin, j+1);
        if(i < end) OrdenedList(i, end);
    }

```

Fonte: código de autoria própria.

A implementação desse código foi feita aplicando o método de ordenação de valores *Quick Sort*, um algoritmo baseado na estratégia de "dividir para conquistar". Esse método consiste em dividir a lista principal em sublistas menores que são ordenadas separadamente, possibilitando, ao final do processo, a ordenação completa da lista.

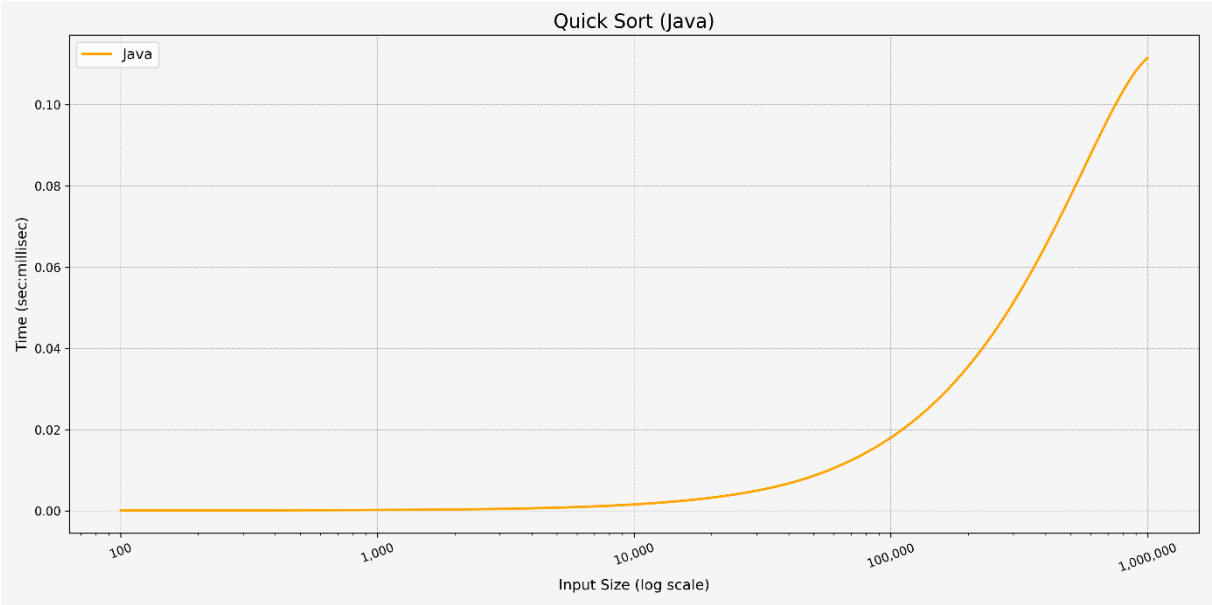
No código apresentado, inicialmente, é criado um método construtor denominado *QuickSort*, que recebe um *array* de valores inteiros como parâmetro. Esse *array*, chamado *ordenedList*, é a estrutura principal onde a ordenação será realizada. Em seguida, um método chamado *OrdenedList* é definido para executar a ordenação do *array*. Dentro desse método, são declaradas quatro variáveis auxiliares: *i*, *j*, *pivo* e *aux*. A variável *i* recebe o valor de *begin*, enquanto *j* recebe o valor de *end - 1*. Já o pivô, atribuído à variável *pivo*, é selecionado como o valor contido no índice central da lista analisada, sendo definido por meio da expressão *ordenedList[(begin + end) / 2]*. A escolha desse pivô tem o objetivo de manter o algoritmo eficiente, uma vez que selecionar um valor central reduz a possibilidade de um pior caso, garantindo um desempenho próximo à complexidade O (n log n).

Na estrutura do código, observa-se a presença de um laço *while* que possui como condição de execução a relação *i* menor ou igual *j*. Esse laço é o principal responsável por reorganizar os elementos da lista ao redor do pivô, assegurando que os valores menores fiquem à esquerda e os maiores à direita. Para isso, dentro do laço principal, há dois laços *while* internos que percorrem os elementos da lista até encontrarem valores que precisem ser trocados. Quando uma troca se faz necessária, o valor contido no índice *i* é armazenado temporariamente na variável *aux*, permitindo que os valores dos índices *i* e *j* sejam permutados entre si. Após essa troca, a variável *i* é incrementada e a variável *j* é decrementada, garantindo a continuidade do processo de partição da lista em torno do pivô.

Concluída essa etapa, o método *OrdenedList* é chamado recursivamente para ordenar as duas partes geradas após a partição inicial. A chamada *OrdenedList(begin, j + 1)* se encarrega da ordenação da sublista à esquerda do pivô, enquanto *OrdenedList(i, end)* realiza a ordenação da sublista à direita. Esse processo de divisão e ordenação ocorre repetidamente até que todas as sublistas estejam organizadas, resultando em um *array* completamente ordenado.

2.5.5. Gráfico de Eficiência em Java

Figura 20: Gráfico de Eficiência do algoritmo *Quick Sort* em Java.



Fonte: Gráfico gerado por algoritmo em linguagem Python.

2.5.6. Análise em Java

O algoritmo *Quick Sort* se destaca por sua eficiência em comparação a outras abordagens clássicas de ordenação, apresentando, em média, uma complexidade O (n log n). Seu desempenho está diretamente relacionado à escolha do pivô, sendo ideal que esse valor permita uma divisão equilibrada da lista, evitando casos em que a recursividade se aproxime de uma complexidade quadrática O (n^2). Por essa razão, a abordagem utilizada nesse código, que seleciona o pivô como o elemento central da lista, contribui para um melhor balanceamento da ordenação e, consequentemente, para um desempenho mais eficiente do algoritmo.

3. CONSIDERAÇÕES FINAIS

Diante da análise supracitada dos algoritmos de ordenação observou-se a favorável e relativa utilização desses algoritmos para a execução da funcionalidade de ordenação em quantidades menos volumosas de dados de entrada, mesmo os algoritmos de complexidade O (n^2), como o *Bubble*, *Insertion* e *Selection Sort*. No entanto, quando é requisitada essa execução de funcionalidade com uma quantidade mais volumosa de dados de entrada, como cem mil e um milhão, os algoritmos menos eficientes de complexidade O (n^2) tornam-se ineficientes para cumprir esse objetivo, tendo destaque apenas os algoritmos *Merge* e *Quick Sort* com, aproximadamente, 99,99% de eficiência se comparado aos outros algoritmos analisados. No entanto, em requisitos de complexidade de implementação os algoritmos analisados de ordem O (n^2) possui maior simplicidade e praticidade do que os algoritmos *Merge* e *Quick Sort*. Por esse motivo, algoritmos como *Bubble*, *Insertion* e *Selection Sort* devem ser pensados como maneiras práticas e menos complexas de utilizar a ordenação de sequências de dados e algoritmos como *Merge* e *Quick Sort* quando é buscado melhor eficiência ou ordenação de uma quantidade volumosa de dados de entrada.

REFERÊNCIAS BIBLIOGRÁFICAS

TARDIVO, Rafael. Bubble sort – Passo a passo. MEDIUM, 2020. Disponível em: https://medium.com/rafaeltardivo/bubble-sort-passo-a-passo-67bb5e11677c. Acesso em: 28 de Janeiro de 2025.

Insertion sort. Wikipedia, 2024. Disponível em: https://en.wikipedia.org/wiki/Insertion_sort. Acesso em: 28 de Janeiro de 2025.

Insertion Sort Algorithm. Geeks for Geeks, 2025. Disponível em: https://www.geeksforgeeks.org/insertion-sort-algorithm/. Acesso em 28 de Janeiro de 2025.

Bubble sort. Wikipedia, 2023. Disponível em: https://pt.wikipedia.org/wiki/Bubble_sort. Acesso em: 11 de fevereiro de 2025.

Insertion sort. Wikipedia, 2023. Disponível em: https://pt.wikipedia.org/wiki/Insertion_sort. Acesso em: 11 de fevereiro de 2025.

Selection sort. Wikipedia, 2024. Disponível em: https://pt.wikipedia.org/wiki/Selection_sort. Acesso em: 11 de fevereiro de 2025.

Merge sort. Wikipedia, 2024. Disponível em: https://pt.wikipedia.org/wiki/Merge_sort. Acesso em: 11 de fevereiro de 2025.

Quicksort. Wikipedia, 2023. Disponível em: https://pt.wikipedia.org/wiki/Quicksort. Acesso em: 11 de fevereiro de 2025.

Bruno. Algoritmos de ordenação: análise e comparação. DEVMEDIA, 2013. Disponível em: https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261. Acesso em: 11 de fevereiro de 2025.

MENDES, Guilherme Ramos. Algoritmos – Merge Sort. MEDIUM, 2021. Disponível em: https://guilherme-rmendes95.medium.com/algoritmos-merge-sort-ef12dadeba2a. Acesso em: 11 de fevereiro de 2025.