UNIVERSIDADE
FEDERAL DO CARIRI

UNIVERSIDADE FEDERAL DO CARIRI
CENTRO DE CIÊNCIAS E TECNOLOGIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Comparação de eficiência entre algoritmos de ordenação clássicos e variações geradas pelo CHAT-GPT

Wanderson Faustino Patricio

Juazeiro do Norte, 30 de outubro de 2023

Sumário

1 Introdução

Ordenar um vetor é muito útil em várias áreas de estudo da computação. Organização dos dados, buscas eficientes, otimização de algritmos e manuseio aprimorado de memória são exemplos onde a utilização de algoritmos de ordenação são extremamente importantes.

Para ordenar um conjunto de dados é necessário comparar as magnitudes dos seus valores e, dependendo do resultado da comparação, reorganizar os dados dentro do conjunto, para que a sequência destes esteja ou de maneira crescente, ou decrescente (no nosso estudo utilizaremos somente ordenação crescente) para que possamos utilizar os dados de maneira eficiente.

Visto a grande utilidade desse tipo de algoritmo em várias áreas de estudo muitos estudiosos da computação pensaram em maneiras de realizar essa tarefa. Bubble sort, selection sort e quick sort são exemplos de algoritmos bastante conhecidos e com análise de complexidade bem conhecida. Todavia, com o intuito de analisar a possibilidade de melhorar a performance desses algoritmos faremos uma análise comparativa entre duas implementações distintas de dois algoritmos bem conhecidos pela comunidade: o Merge Sort e o Insertion Sort.

Para esse experimento foi fornecido ao Chat-GPT um exemplo de implementação de cada um dos algoritmos e pedido para que ele realizasse mudanças que aprimorassem o desempenho dos mesmos. Com ambos os códigos em mão faremos uma análise de complexidade de maneira matemática (tendo como referência a quantidade de comparações que cada algoritmo realiza) e empírica (a partir do tempo de execução de cada um para um mesmo conjunto de dados) e compararemos se estas mudanças realmente melhoram o desempenho dos algoritmos.

2 Fundamentação Teórica

2.1 Merge Sort

O algoritmo de ordenação Merge Sort é um método eficiente e de divisão e conquista para ordenar listas ou arrays. O seu funcionamentos se baseia em:

i) Divisão: A primeira etapa do Merge Sort consiste em dividir a lista não ordenada em duas metades iguais. Isso é feito recursivamente em cada nova metade até cada sublista conter apenas um elemento (nosso caso base), pois listas de um elemento são consideradas ordenadas.

ii) Conquista: Após a divisão das sublistas, teremos vários vetores menores já ordenados, sendo, portanto, necessário apenas unir novamente os subvetores.

iii) Mesclagem (merge): Durante a mesclagem, os elementos são comparados nas sublistas e movidos para a lista de saída em ordem crescente. O menor elemento de ambas as sublistas é escolhido e movido para a lista de saída. Esse processo continua até que todas as sublistas tenham sido completamente mescladas.

iv) Recursividade: A etapa de mesclagem é realizada de forma recursiva. Isso significa que as sublistas menores são mescladas primeiro e, em seguida, suas saídas são mescladas, criando uma árvore de chamadas recursivas. O processo continua até que todas as sublistas tenham sido mescladas em uma única lista ordenada.

v) Ordenação Completa: Após todas as sublistas estarem ordenadas teremos o vetor original ordenado.

2.1.1 Original

Inicialmente, apresentaremos o algoritmo clássico, que foi apresentado ao Chat-GPT para ser melhorado.

```
void mergeOriginal(int* vet, int inicio, int meio, int fim){
    int *temp, p1, p2, tamanho, i, j, k;
    int fim1 = 0, fim2 = 0;
    tamanho = fim-inicio +1;
    p1 = inicio;
    p2 = meio + 1;
    temp = (int*) malloc(tamanho * sizeof(int));
    if(temp != NULL){
        for(i = 0; i < tamanho; i++){
            if(!fim1 && !fim2){
                if(vet[p1]< vet[p2]){
                    temp[i] = vet[p1];
                    p1++;
                }else{
                    temp[i] = vet[p2];
                    p2++;
                }
                if(p1>meio){
                    fim1 = 1;
                }
                if(p2>fim){
                    fim2 = 1;
                }
            }else{
                if(!fim1){
```

```
26                      temp[i] = vet[p1];
27                      p1++;
28                  }else{
29                      temp[i] = vet[p2];
30                      p2++;
31                  }
32              }
33          }
34          for(j = 0, k = inicio; j < tamanho; j++, k++){
35              vet[k] = temp[j];
36          }
37      }
38      free(temp);
39  }
40
41  void mergeSortOriginal(int *vet, int inicio, int fim){
42      int meio;
43      if(inicio < fim){
44          meio = floor((inicio + fim)/2);
45          mergeSortOriginal(vet, inicio, meio);
46          mergeSortOriginal(vet, meio+1, fim);
47          mergeOriginal(vet, inicio, meio, fim);
48      }
49  }
```

2.1.2 Análise assintótica de complexidade do algoritmo original

Consideremos que cada comparação demore um tempo c para ser executada e que cada atribuição demore um tempo a para se executada. As outras ações podem ser consideradas desprezíveis em relação aos dois acima.

para a mesclagem de um vetor de tamanho s ($s = fim - inicio + 1$) temos os seguintes passos:

1. Linhas 2 a 6: 5 atribuições ($5a$).
2. Linha 8: comparação (c).
3. Para cada iteração nas linhas de 10 a 32 ao máximo 4 comparações e 4 atribuições ($4c + 4a$). Como ao todo há s iterações ($4s(a + c)$).
4. Linhas 34 a 36: s iterações, cada uma com uma atribuição ($s \cdot a$).

Portanto, o tempo de execução do algoritmo de merge em um vetor de tamanho s é:

$$T_m(s) = (5a + c) + (5a + 4c) \cdot s$$

Para o algoritmo MergeSort teremos que ele será executado em ambas as metades, e após isso será executado o algoritmo de merge no vetor de tamanho n (tamanho total). Sendo $T(n)$ o tempo de execução do merge sort temos

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + T_m(n)$$

Recursivamente teremos

1. $$T(n) = 2\left[2 \cdot T\left(\frac{n}{4}\right) + T_m\left(\frac{n}{2}\right)\right] + T_m(n) = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2 \cdot T_m\left(\frac{n}{2}\right) + T_m(n)$$

2.

$$T(n) = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 4 \cdot T_m\left(\frac{n}{4}\right) + 2 \cdot T_m\left(\frac{n}{2}\right) + T_m(n)$$

3. Por indução

$$T(n) = 2^p \cdot T\left(\frac{n}{2^p}\right) + \sum_{k=1}^{p} 2^k \cdot T_m\left(\frac{n}{2^k}\right)$$

Considerando que para o caso em que $n = 1$ teremos que fazer apenas uma comparação, $T(1) = c$. Logo, tomemos $p = \log n$

$$T(n) = 2^{\log n} \cdot T\left(\frac{n}{2^{\log n}}\right) + \sum_{k=1}^{\log n} 2^k \cdot T_m\left(\frac{n}{2^k}\right)$$

$$T(n) = n \cdot T(1) + \sum_{k=1}^{\log n} 2^k \cdot \left((5a + c) + (5a + 4c) \cdot \frac{n}{2^k}\right)$$

$$T(n) = c \cdot n + (5a + c) \sum_{k=1}^{\log n} 2^k + (5a + 4c)n \sum_{k=1}^{\log n} 1$$

$$T(n) = c \cdot n + (5a + c) \cdot 2 \cdot \frac{2^{\log n + 1} - 1}{2 - 1} + (5a + 4c) \cdot n \cdot (\log n - 1)$$

Simplificando temos

$$\boxed{T(n) = (5a + 4c) \cdot n \cdot \log n + (15a + c) \cdot n - (10a + 2c)} \tag{1}$$

Pela análise assintótica (big O) a complexidade será

$$\boxed{T(n) \in O(n \cdot \log n)} \tag{2}$$

2.1.3 Código do Chat-GPT

Ao apresentar o código do merge sort acima a inteligência artificial da openai apontou os seguintes pontos que poderiam ser ajustados.

- "Otimizar alocação de memória: em vez de alocar memória para o array temporário dentro da função de mesclagem, você pode alocá-lo uma vez na função mergeSort. Isso reduzirá a sobrecarga de alocações frequentes de memória e desalocações. Além disso, certifique-se de liberar a memória alocada após a classificação."

Esse ponto é relevante para o tempo, visto que, mesmo não demorando muito tempo para alocar espaço na memória, ao repetir esse processo várias vezes dentro da árvore de recursão o compilador terá que varrer a memória para procurar espaços vazios.

- "Evite recursão para subvetores pequenos: Para subvetoress muito pequenos, a recursão pode adicionar sobrecarga. Você pode definir um limite abaixo do qual o algoritmo usa um algoritmo de classificação mais simples, como classificação por inserção. Isso pode ser mais rápido para vetores pequenos."

Durante a execução teremos muitos vetores pequenos a ser ordenados. Evitando entrar em recursão e executando algoritmos recursivos para estes subvetores teremos execuções mais rápidas.

Apresentando o código apresentado pela IA temos

```c
void mergeGPT(int* vet, int inicio, int meio, int fim) {
    int tamanho = fim - inicio + 1;
    int* temp = (int*)malloc(tamanho * sizeof(int));
    if (temp != NULL) {
        int p1 = inicio;
        int p2 = meio + 1;
        for (int i = 0; i < tamanho; i++) {
            if (p1 <= meio && (p2 > fim || vet[p1] <= vet[p2])) {
                temp[i] = vet[p1];
                p1++;
            } else {
                temp[i] = vet[p2];
                p2++;
            }
        }
        for (int j = 0, k = inicio; j < tamanho; j++, k++) {
            vet[k] = temp[j];
        }
        free(temp);
    }
}

void mergeSortGPT(int* vet, int inicio, int fim) {
    if (inicio < fim) {
        if (fim - inicio < 5) {
            // Use insertion sort for small arrays
            for (int i = inicio; i <= fim; i++) {
                int key = vet[i];
                int j = i - 1;
                while (j >= inicio && vet[j] > key) {
                    vet[j + 1] = vet[j];
                    j--;
                }
                vet[j + 1] = key;
            }
        } else {
            int meio = inicio + (fim - inicio) / 2;
            mergeSortGPT(vet, inicio, meio);
            mergeSortGPT(vet, meio + 1, fim);
            mergeGPT(vet, inicio, meio, fim);
        }
    }
}
```

2.1.4 Análise assintótica de complexidade do algoritmo do Chat-GPT

Para o algoritmo de merge em um vetor de tamanho s teremos

1. Linha 2: uma atribuição (a)
2. Linha 4: uma comparação (c)
3. Linhas 5 e 6: 2 atribuições ($2a$)
4. Para cada iteração (linhas 8 a 13): 3 comparações e 2 atribuições ($3c + 2a$)
5. s iterações ($s \cdot (3c + 2a)$)
6. Linhas 16 a 18: s iterações com uma atribuição ($s \cdot a$))

O tempo para o merge será

$$T_m(s) = (3a + c) + 3(a + c) \cdot s$$

Análogo ao código original teremos

$$T(n) = 2^p \cdot T\left(\frac{n}{2^p}\right) + \sum_{k=1}^{p} 2^k \cdot T_m\left(\frac{n}{2^k}\right) \tag{3}$$

Considerando o caso para $n = 5$ só será executado o algoritmo insertion sort. Neste algoritmo ocorrem 5 iterações, cada uma com 3 atribuições obrigatórias e mais 2 possivéis para o loop while. Portanto

$$T(5) = \sum_{k=1}^{5}(3a + (k-1) \cdot 2a) = \sum_{k=1}^{5}(a + k \cdot a)$$

$$T(5) = 5a + 15a = 20a$$

Aplicando $p = \log\left(\frac{n}{5}\right)$ na equação (3) teremos

$$T(n) = 2^{\log\left(\frac{n}{5}\right)} \cdot T\left(\frac{n}{2^{\log\left(\frac{n}{5}\right)}}\right) + \sum_{k=1}^{\log\left(\frac{n}{5}\right)} 2^k \cdot T\left(\frac{n}{2^k}\right)$$

$$T(n) = \frac{n}{5} \cdot T(5) + \sum_{k=1}^{\log\left(\frac{n}{5}\right)} 2^k \cdot \left[(3a + c) + 3(a + c)\frac{n}{2^k}\right]$$

$$T(n) = 4a \cdot n + (3a + c) \sum_{k=1}^{\log\left(\frac{n}{5}\right)} 2^k + 3(a + c)n \sum_{k=1}^{\log\left(\frac{n}{5}\right)} 1$$

$$T(n) = 4a \cdot n + (3a + c) \cdot 2 \cdot \frac{\frac{2n}{5} - 1}{2 - 1} + 3(a + c) \cdot n \cdot (\log n - \log 5 - 1)$$

Portanto

$$\boxed{T(n) = (3a + 3c) \cdot n \cdot \log n + \left(\frac{14a + 4c}{5}\right) \cdot n + (3a - c) - 3(\log 5)(a + c)} \tag{4}$$

Através da análise assintótica

$$\boxed{T(n) \in O(n \cdot \log n)} \tag{5}$$

Comparando as análises assintóticas não há diferença nas complexidades dos algoritmos, pois ambos são $O(n \log n)$, porém, o termo que acompanha o componente $n \cdot \log n$ é menor no algoritmo gerado pelo Chat-GPT, o que fará uma diferença significativa para valores altos de n.

A partir da análise matemática da quantidade de comparações e atribuições vemos que o código gerado pela inteligência artificial é mais eficiente.

2.2 Insertion Sort

O Insertion Sort, ou "ordenação por inserção"em português, é um algoritmo de ordenação simples e eficiente. Ele funciona da seguinte forma:

1. Divisão da Lista: Considerando que uma parte da lista já está ordenada podemos dividi-la em duas partes: a ordenada e a não ordenada. A parte ordenada começa vazia, e a parte não ordenada contém todos os elementos.

2. Inserção: O algoritmo percorre a parte não ordenada da lista um elemento de cada vez. Ele remove um elemento da parte não ordenada e o insere na posição correta da parte ordenada. Isso é feito comparando o elemento atual com os elementos na parte ordenada e deslocando os elementos na parte ordenada, se necessário, para abrir espaço para o novo elemento.

3. Repetição: O passo de inserção é repetido até que todos os elementos da parte não ordenada tenham sido movidos para a parte ordenada. No final, a lista estará completamente ordenada.

O Insertion Sort é eficiente para listas pequenas ou quase ordenadas, mas ineficiente em listas muito longas, devido ao seu tempo de execução no pior caso que é $O(n^2)$. No entanto, ele tem a vantagem de ser in-place, ou seja, não requer memória adicional para ordenação, e estável, mantendo a ordem relativa de elementos com chaves iguais. É frequentemente usado como um passo intermediário em algoritmos de ordenação mais complexos, como o Merge Sort e o Quick Sort.

2.2.1 Original

```
1  void insertionSortOriginal(int *vet, int n){
2      int i, j, aux;
3      for(i = 1; i < n; i++){
4          aux = vet[i];
5          j = i;
6          while(j > 0 && aux < vet[j-1]){
7              vet[j] = vet[j-1];
8              j--;
9          }
10         vet[j] = aux;
11     }
12 }
```

2.2.2 Análise assintótica de complexidade do algoritmo original

Ocorrerão $n-1$ iterações no algoritmo.
Para cada iteração i há:

1. Linhas 4 e 5: 2 atribuições $(2a)$.

2. No máximo $(n-i-1)$ iterações no loop while, cada uma com 2 atribuições e 2 comparações $((n-i)\cdot(2a+2c))$.

3. Linha 10: 1 atribuição (a).

Logo, o tempo total será:

$$T(n) = \sum_{i=1}^{n-1}(3a + (n-i-1)\cdot(2a+2c))$$

$$T(n) = (a - 2c + (2a + 2c)n) \cdot \sum_{i=1}^{n-1} 1 - (2a + 2c) \cdot \sum_{i=1}^{n-1} i$$

$$T(n) = (a - 2c + (2a + 2c)n)(n - 1) - (a + c)n(n - 1)$$

Portanto

$$\boxed{T(n) = (a + c) \cdot n^2 - 3c \cdot n + (2c - a)} \tag{6}$$

Pela análise assintótica

$$\boxed{T(n) \in O(n^2)} \tag{7}$$

2.2.3 Código do Chat-GPT

Os comentários feitos pela inteligência artificial foram as seguintes:

- "Pesquisa binária pelo ponto de inserção: em vez de deslocar os elementos um por um para encontrar o ponto de inserção, você pode usar uma pesquisa binária para localizar a posição correta do elemento atual. Isso reduzirá o número de comparações no loop interno."

Com a pesquisa binária pelo ponto correto de inserção reduziremos um tempo de procura que é linear no vetor para um tempo de ordem logarítmica, o que apresentará uma redução bem significativa.

- "Use uma variável temporária: No loop interno, use uma variável temporária para armazenar o elemento atual que você está tentando inserir, para apenas fazer uma única atribuição ao array depois de encontrar a posição correta. Isso reduz o número de atribuições."

ao invés de fazermos vária atribuições até chegarmos a posição correta, podemos armazenar a posição atual até chegarmos a correta.

```
1  void insertionSortGPT(int *vet, int n) {
2      for (int i = 1; i < n; i++) {
3          int temp = vet[i];
4          int low = 0, high = i - 1;
5          while (low <= high) {
6              int mid = (low + high) / 2;
7              if (temp < vet[mid]) {
8                  high = mid - 1;
9              } else {
10                 low = mid + 1;
11             }
12         }
13
14         for (int j = i; j > low; j--) {
15             vet[j] = vet[j - 1];
16         }
17         vet[low] = temp;
18     }
19 }
```

2.2.4 Análise assintótica de complexidade do algoritmo do Chat-GPT

Para cada iteração for teremos:

1. Linhas 3 e 4: 3 atribuições ($3a$)

2. Linhas 5 a 12: Para cada loop while 2 comparações e duas atribuições $(\log(i) \cdot (2a + 2c))$

3. Linhas 14 a 16: se as comparações do loop while levar para uma posição próxima a i o segundo loop for será executado em $O(1)$, caso o vetor já esteja quase ordenado. caso contrário o loop for será executado em $O(n-i)$ o que não diferenciará do algoritmo previamente apresentado. Portanto, consideraremos que será executada a mesclagem entre primeira opção. Ao todo teremos 1 atribuição (a)

4. Linha 17: 1 atribuição (a)

O tempo total será

$$T(n) = \sum_{i=1}^{n-1} \left(5a + \frac{1}{2}(2a + 2c)\log(i) + \frac{1}{2} \cdot a \cdot (n - i) \right)$$

$$T(n) = 5a(n-1) + \frac{an(n-1)}{4} + (a + c)\sum_{i=1}^{n-1} \log(i)$$

$$\boxed{T(n) = \frac{a}{4} \cdot n^2 + \frac{19a}{4} \cdot n - 5a + (a + c)\sum_{i=1}^{n-1} \log(i)} \tag{8}$$

A análise assintótica conduz a

$$\boxed{T(n) \in O(n^2)} \tag{9}$$

Apesar da soma logarítmica não ser facil de calcular, ela é da ordem de $O(\log n)$ o que indica a redução do tempo. Ademais, o coeficiente que acompanha n^2 é pelo menos 4 vezes menor que o do algoritmo original, indicando que este algoritmo é mais eficiente.

3 Objetivos

Durante este experimento iremos tentar responder as seguintes perguntas:

3.1 Gerais

i) A análise matemática da função de complexidade dos algoritmos está condizente com os resultados experimentais?

ii) Há diferenças significativas entre o tempo de ordenação para valores altos de tamanhos de arrays?

iii) Os algoritmos gerados pelo Chat-GPT apresentam desempenho melhorado em relação às implementações clássicas?

3.2 Específicos

i) Como saber quais fatores realmente são importantes para a melhora do desempenho de um algoritmo?

ii) De que forma a análise assintótica pode influenciar os programadores a escolher algoritmos que não são realmente eficientes?

4 Procedimento Experimental

Para verificar se realmente há uma diferença no tempo de execução nos algoritmos foi criado um programa que gerava 10 vetores, os quais eram preenchidos com valores aleatórios e postos em ambos os algoritmos de ordenação. Para evitar que a disposição dos valores alterasse o tempo de execução, os mesmos valores (na mesma ordem) foram utilizados para a ordenação.

Para comparar se o tamamanho do vetor influenciaria no resultado foram ordenados vetores de 5 tamanhos diferentes (10, 100, 1000, 10000 e 100000) os quais foram comparados caso a caso.

4.1 Apresentação do algoritmo principal

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "functions.h"

void preenche_vet(int* vet, int n, int maximo){
    for(int i=0; i<n; i++){
        vet[i] = rand()%maximo + 1;
    }
}

void copia_vet(int* vet1, int* vet2, int n){
    for(int i=0; i<n; i++){
        vet2[i] = vet1[i];
    }
}

int main(){
    srand(time(NULL));

    int max_n = 5;

    int n, tam = pow(10,max_n), maximo = 1000000;
    int vet[tam], vet_aux[tam];

    clock_t start, end;
    double cpu_time_used;

    printf("MERGE SORT \n");

    for(int j = 1; j<=max_n; j++){
        n = pow(10, j);
        printf("Experimento com vetores de tamanho %d: \n\n", n);

        for(int i = 0; i<10; i++){
            preenche_vet(vet, n, maximo);

            // Original
            copia_vet(vet, vet_aux, n);
            start = clock();
            mergeSortOriginal(vet_aux, 0, n-1);
            end = clock();
            cpu_time_used = (double) (end-start)/CLOCKS_PER_SEC;
            printf("Experimento %d ::: Original: %.4f s --- ",i+1,
    cpu_time_used);

```

```
            // Chat-GPT
            copia_vet(vet, vet_aux, n);
            start = clock();
            mergeSortGPT(vet_aux, 0, n-1);
            end = clock();
            cpu_time_used = (double) (end-start)/CLOCKS_PER_SEC;
            printf("GPT: %.4f s \n", cpu_time_used);
        }

        printf("\n\n");
    }

    printf("INSERTION SORT \n");

    for(int j = 1; j<=max_n; j++){
        n = pow(10, j);
        printf("Experimento com vetores de tamanho %d: \n\n", n);

        for(int i = 0; i<10; i++){
            preenche_vet(vet, n, maximo);

            // Original
            copia_vet(vet, vet_aux, n);
            start = clock();
            insertionSortOriginal(vet_aux, n);
            end = clock();
            cpu_time_used = (double) (end-start)/CLOCKS_PER_SEC;
            printf("Experimento %d ::: Original: %.4f s --- ",i+1,
    cpu_time_used);

            // Chat-GPT
            copia_vet(vet, vet_aux, n);
            start = clock();
            insertionSortGPT(vet_aux, n);
            end = clock();
            cpu_time_used = (double) (end-start)/CLOCKS_PER_SEC;
            printf("GPT: %.4f s \n", cpu_time_used);
        }

        printf("\n\n");
    }

    return 0;
}
```

Há algumas observações a serem feitas a respeito do programa principal de teste:

a) Foi criado a função preenche_vet(int* *vet*, int *n*, int *maximo*), que preenche *n* posições de um vetor *vet* com valores aleatórios variando de 1 até um valor *maximo*. Essa função tem complexidade $O(n)$.

b) A função copia_vet(int* *vet*1, int* *vet*2, int *n*) copia as n primeiras posições de um vetor *vet*1 para um vetor *vet*2 (Essa função garante que em ambos os algoritmos de ordenação utilizaremos os mesmos vetores). Esse algoritmo tem complexidade $O(n)$.

Embora essas funções serão utilizadas em todos os testes, o que resultaria em uma diferença no tempo de execução dos algoritmos, o tempo só será contabilizado após a variável start receber o clock(). Logo, o efeito será comepensado nos testes.

Para medir o tempo de execução do código fazemos a variável *start* receber o clock do computador no momento antes do algoritmo ser executado e a variável *end* receber o clock imediatamente

após o fim da ordenação. Quando o *start* e o *end* são contabilizados calculamos a sua diferença dividida pela quantidade de clocks por segundo (CLOCKS_PER_SEC).

Para cada tamanho de vetor foram realizados 10 testes, cada um variando os elementos nos vetores através da função preenche_vet. O experimento é realizado tanto para o mergeSort quanto para o insertionSort.

4.2 Resultados do experimento

Merge Sort

Tamanho dos vetores									
n=10		n=100		n=1000		n=10000		n=100000	
original	GPT	original	GPT	original	GPT	original	GPT	original	GPT
0,0000	0,0000	0,0000	0,0000	0,0000	0,0010	0,0090	0,0050	0,0870	0,0600
0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0080	0,0050	0,0840	0,0600
0,0000	0,0000	0,0000	0,0000	0,0010	0,0010	0,0080	0,0050	0,0830	0,0600
0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0080	0,0050	0,0840	0,0590
0,0000	0,0000	0,0000	0,0000	0,0010	0,0010	0,0080	0,0040	0,0830	0,0590
0,0000	0,0000	0,0000	0,0000	0,0010	0,0010	0,0070	0,0040	0,0840	0,0600
0,0000	0,0000	0,0010	0,0000	0,0010	0,0000	0,0080	0,0040	0,0830	0,0600
0,0000	0,0000	0,0000	0,0000	0,0010	0,0000	0,0080	0,0040	0,0840	0,0590
0,0000	0,0000	0,0000	0,0000	0,0010	0,0000	0,0080	0,0050	0,0830	0,0600
0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0080	0,0040	0,0820	0,0600

Tabela 1: Resultados do tempo de execução (em s) do Merge Sort

Insertion Sort

Tamanho dos vetores									
n=10		n=100		n=1000		n=10000		n=100000	
original	GPT	original	GPT	original	GPT	original	GPT	original	GPT
0,0000	0,0000	0,0000	0,0000	0,0030	0,0020	0,3350	0,1930	33,8350	19,4460
0,0000	0,0000	0,0000	0,0000	0,0040	0,0020	0,3390	0,1960	33,7680	19,3950
0,0000	0,0000	0,0001	0,0000	0,0030	0,0020	0,3400	0,1960	33,8750	19,4800
0,0000	0,0000	0,0000	0,0000	0,0030	0,0020	0,3380	0,1930	33,7060	19,3600
0,0000	0,0000	0,0000	0,0000	0,0040	0,0020	0,3390	0,1960	33,8870	19,4410
0,0000	0,0000	0,0000	0,0000	0,0030	0,0020	0,3410	0,1950	33,6530	19,3590
0,0000	0,0000	0,0010	0,0000	0,0030	0,0020	0,3430	0,1980	33,8780	19,4730
0,0000	0,0000	0,0000	0,0000	0,0030	0,0020	0,3340	0,1930	33,8480	19,4390
0,0000	0,0000	0,0000	0,0000	0,0030	0,0030	0,3410	0,1970	33,6970	19,3790
0,0000	0,0000	0,0001	0,0000	0,0030	0,0020	0,3360	0,1920	33,7760	19,4030

Tabela 2: Resultados do tempo de execução (em s) do Insertion Sort

As tabelas 1 e 2 apresentam o resultados dos tempos de execução dos algoritmos (executados de forma pareada) para os diferentes valores do tamanho dos vetores.

Inicialmente, definiremos as métricas que utilizaremos para a análise de eficiência: a média entre os tempos de execução e a diferença entre o maior e o menor tempo de execução dos algoritmos.

Médias dos tempos

Merge Sort									
Tamanho dos vetores									
n=10		n=100		n=1000		n=10000		n=100000	
original	GPT	original	GPT	original	GPT	original	GPT	original	GPT
0,0000	0,0000	0,0001	0,0000	0,0006	0,0004	0,0080	0,0045	0,0837	0,0600
Insertion Sort									
Tamanho dos vetores									
n=10		n=100		n=1000		n=10000		n=100000	
original	GPT	original	GPT	original	GPT	original	GPT	original	GPT
0,0000	0,0000	0,0012	0,0000	0,0032	0,0020	0,3386	0,1949	33,7923	19,4175

Tabela 3: Médias dos tempos de execução (em s)

Diferenças nos tempos

Merge Sort									
Tamanho dos vetores									
n=10		n=100		n=1000		n=10000		n=100000	
original	GPT	original	GPT	original	GPT	original	GPT	original	GPT
0,0000	0,0000	0,0010	0,0000	0,0010	0,0010	0,0020	0,0010	0,0050	0,0010
Insertion Sort									
Tamanho dos vetores									
n=10		n=100		n=1000		n=10000		n=100000	
original	GPT	original	GPT	original	GPT	original	GPT	original	GPT
0,0000	0,0000	0,0010	0,0000	0,0010	0,0010	0,0080	0,0060	0,234	0,121

Tabela 4: diferença entre os maiores e os menores tempos de execução (em s)

4.3 Análise dos resultados

4.3.1 Análise estatística

Ao fazermos uma análise estatística considerando os tempos médios de execução para os tempos médios de execução podemos encontrar os coeficiente de correlação de Pearson para ver se há uma correlação entre o tempo e o tamanho do vetor.

a) **Merge Sort Original:** ao fazer a regressão linear com o tempo em função de $n \cdot \log n$ encontramos uma correlação de $99,9\%$.

b) **Merge Sort modificado:** da mesma maneira do item a), encontramos uma correlação de $93,9\%$.

c) **Insertion Sort Original:** ao fazer a regressão linear com o tempo em função de n^2 encontramos uma correlação de 99.9%.

d) **Insertion Sort modificado:** da mesma maneira do item c), encontramos uma correlação de $99,9\%$.

Através da análise de regressão podemos perceber que a predição matemática através das comparações e atribuições (análise assintótica) está condizente com o resultado dos experimentos. Esses resultados indicam que mesmo sem executar realmetne o programa podemos ter a noção de como ele irá se comportar.

4.3.2 Comparação entre os algoritmos

Ao olhar para os tempos médios (tabela 3) vê-se que há uma diferença entre os tempos de execução. Os algoritmos de ordenação gerados pelo Chat-GPT apresentaram um desmpenho melhor em relação a implementação clássica.

Todavia, é interessante notar que o algoritmo Merge Sort não apresentou uma melhora tão significativa, visto que, mesmo para valores exuberantes ($n > 10^5$) de tamanhos de vetores, o tempo de execução não passa de $0.1\,s$. Porém, mesmo nesses casos, há uma redução não desprezível no tempo de execução.

Para o Insertion Sort, entretanto, a diferença foi mais evidente. A partir de $n = 1000$ o desempenho do algoritmo gerado pela IA apresentou-se quase 2 vezes maior que o algoritmo clássico. Mesmo para o caso $n = 10^5$, em que o tempo de execução do código clássico é $33\,s$, as mudanças do Chat-GPT conseguiram reduzir para menos de $30\,s$. Desta forma, vemos que mesmo pequenas mudanças podem provocar mudanças relevantes em grandes escalas.

Olhando para as diferenças entre os tempos máximo e mínimo percebemos também que há uma consistência nos procedimentos, pois não há discrepância entre execuções em situações similiares, com a diferença não excedendo 5%.

5 Discussão dos resultados

Após realizado os experimentos podemos chegar as seguintes conclusões:

- A análise assintótica dos algoritmos feita previamente no início do artigo (2^a seção) está de acordo com os resultados experimentais de tempo, estudados por uma análise estatística.

  O código do Merge Sort leva a uma complexidade $O(n \cdot \log n)$. Como a regressão linear ajustada revelou um correlação linear de Pearson maior que 90% há um indício bastante forte de que para $n \to \infty$, $T(n) \to C \cdot n \cdot \log n$ ($C > 0$ uma constante).

  De forma análoga, temos que o código do Insertion Sort indica uma tendência quadrática ($T(n) = O(n^2)$). Interessante notar que a ordenação por inserção realmente apresenta desempenho inferior ao de merge, sendo o tempo de execução, para $n = 10^5$, 400 vezes maior.

- O algoritmo Merge Sort não apresenta grandes diferenças de tempo, sua execução é concisa, demorando tempos relativamente próximos uns dos outros. É também um algoritmo estável, pois, de acordo com a tabela 4 vemos que não diferenças entre os valores máximo e mínimo de tempo para um mesmo tamanho de array.

  Para o Insertion Sort, porém, vemos que quanto maiores os tamanhos dos vetores há diferenças exorbitantes no tempo de execução, excedendo $30\,s$. O código porém é bastante estável, não tendo diferenças grandes entre o tempo mínimo e máximo.

- Para arrays grandes os códigos gerados pelo Chat-GPT indicaram tempos bem melhores. mesmo o algoritmo de merge não excedendo $0,1\,s$ há uma diferença de $0,02\,s$. Todavia, há uma grande diferença no algoritmo de inserção, chegando a ser $14\,s$ mais rápido.

  Para arrays de tamnho pequeno, entretanto, não há diferenças significativas entre os códigos, ficando a cargo do programador escolher qual algoritmo utilizar.

- Fatores relevantes para a execução de um programa podem variar de processador para processador, porém, há alguns fatores que são intrínsecos ao funcionamento do código, tais como o tempo de atribuição de uma variável a outra, o tempo de comparação, o tempo para somar duas variáveis, o espaço disponível na memória (mais relevantes em aplicações embarcadas), etc. Esses fatores devem ser levados em conta pelo programador na hora de montar sua aplicação).

- A análise assintótica dos algoritmos pode levar a uma excelente comparação entre algoritmos com fundamentações distintas, pois para vetores grandes os tempos de execução condizem com os valores experimentais.

  Todavia, a prática revela se a sua aplicação vai cumprir com o esperado, podendo ser escolhidos pelos programadores códigos que não são eficientes.

6 Conclusão

Ao final do experimento podemos concluir que a análise de algoritmos para a sua compreensão é uma excelente maneira de os programadores entenderem o funcionamento de seus códigos. Todavia, a análise pode não ser fiel a execução desses algoritmos na prática, sendo necessário escolher parâmetros de mudança para melhorar a execução dos mesmos.

Através de comparaçções exeperimentais foi relevado que alterações feitas pelo Chat-GPT no código clássico a este geraram desempenhos melhores para todos os casos de teste a que foram submetidos, mesmo, algumas vezes, não sendo uma mudança perceptível em aplicações cotidianas.

Tendo isso em mente, podemos considerar que a utilização de ferramentas de processamento de linguagem natural para auxílio na criação de aplicações é um excelente recurso desde que acompanhado pelo bom senso do programador. Não se pode esperar que aplicações completas sejam feitas por essas ferramentas nem que não ocorram erros nas suas implementações, ficando a cargo do programador escolher o que utilizar nas suas aplicações e como fazer sua implementação.