Nome: Leonardo Cremasco Bernardes Boscariol
RA: 202310228

Ex.2) Vetor de 100 elementos e suposições.

1) o vetor está sempre cheio

```
#include <stdio.h>

int find_min_non_recursive(int arr[], int length) {
    int min_value = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }
    return min_value;
}
```

*Complexidade: ***Tempo****: O algoritmo percorre todos os 100 elementos uma vez, então a complexidade é O(n), onde 'n' é o número de elementos (100).* ***Espaço****: A complexidade de espaço é O(1), já que o algoritmo usa um número fixo de variáveis auxiliares(tempo e espaço constantes).*

2) o vetor não está sempre cheio

```
int find_min_non_recursive(int arr[], int length) {
    int min_value = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }
    return min_value;
}
```

*Complexidade: ***Tempo****: O algoritmo percorre todos os 100 elementos uma vez, então a complexidade é O(n), onde 'n' é o número de elementos (100).* ***Espaço****: A complexidade de espaço é O(1), já que o algoritmo usa um número fixo de variáveis auxiliares; (tempo linear com o número de elementos presentes).*

3) o vetor está ordenado

```
int find_min_non_recursive(int arr[], int size) {
    return arr[0];
}
```

Complexidade: Tempo*: A complexidade é O(1), pois só precisa acessar o primeiro elemento.*
Espaço*: A complexidade de espaço é O(1);* (*acesso constante ao menor elemento).*

4) o vetor não está ordenado

```
int find_min_non_recursive(int arr[], int length) {
    int min_value = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }
    return min_value;
}
```

***Complexidade: Tempo**: O(n).**Espaço**: O(1);(tempo linear).*

5) O vetor não é recursivo

```
#include <stdio.h>
#include <limits.h>

int encontrar_menor_valor_iterativo(int vetor[], int tamanho) {
    int menor_valor = INT_MAX;  // para começar c o maior valor inteiro
possivel
    for (int i = 0; i < tamanho; i++) {
        if (vetor[i] < menor_valor) {
            menor_valor = vetor[i];
        }
    }
    return menor_valor;
}
```

***Complexidade Iterativa(não recursiva)**: O(n)— o algoritmo precisa iterar sobre todos os 100 elementos para encontrar o menor valor.*

6) o vetor é recursivo

```
#include <stdio.h>

int encontrar_menor_valor_recursivo(int vetor[], int n) {
    if (n == 1) {
        return vetor[0];
    } else {
        int menor_valor_restante = encontrar_menor_valor_recursivo(vetor, n-1);
        return (vetor[n-1] < menor_valor_restante) ? vetor[n-1] :
menor_valor_restante;
    }
}
```

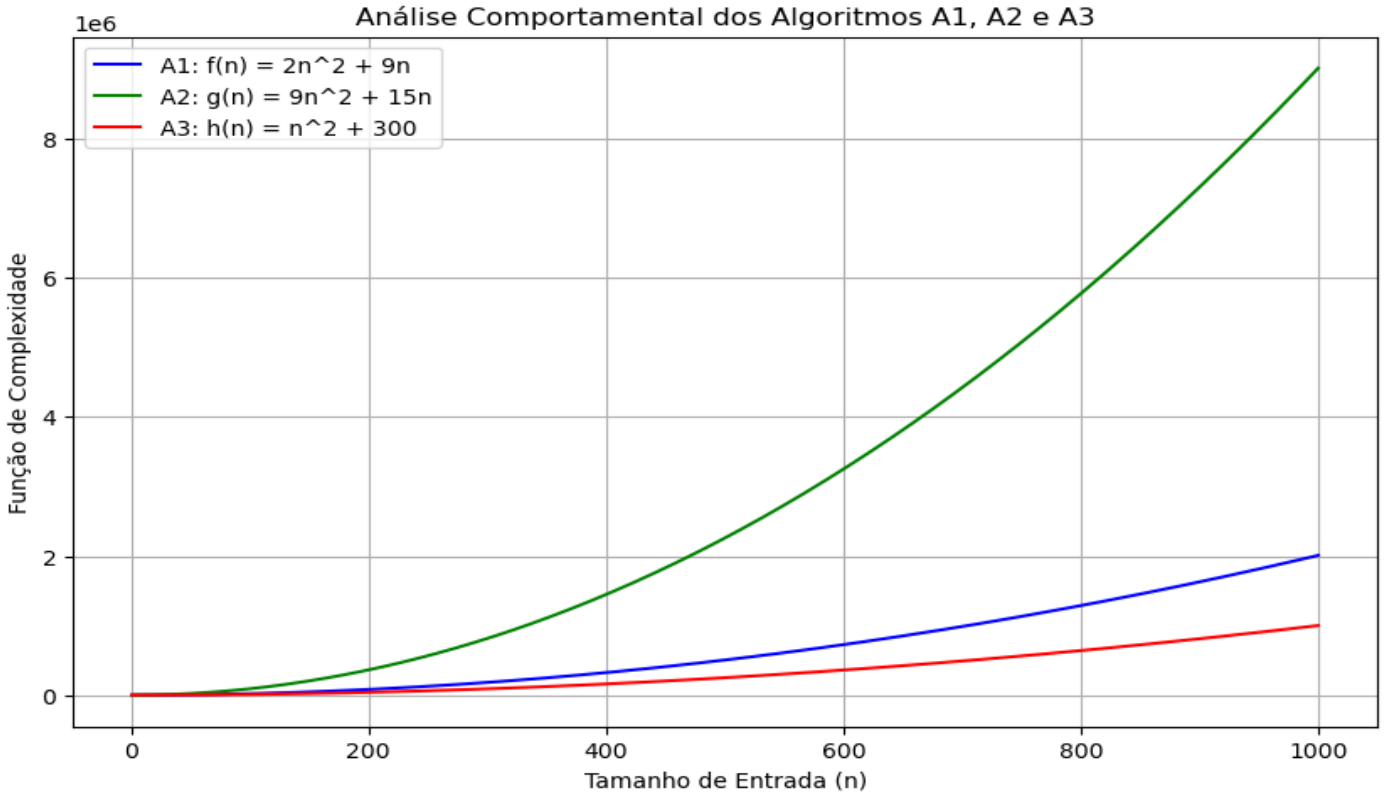
***Complexidade:** cada chamada recursiva reduz o tamanho do vetor em 1, até atingir o caso base. A profundidade da recursão será de 'n' (100 neste caso); espaço linear em relação ao número de elementos devido à recursão.*

Ex.3) Gráficos dos Algoritmos em O (n²)

Python 3.12

```
import numpy as np
import matplotlib.pyplot as plt

# definindo a faixa de valores para n
n = np.linspace(1, 1000, 500)

# funções de complexidade dos algoritmos do ex.
f_n = 2 * n**2 + 9 * n
g_n = 9 * n**2 + 15 * n
h_n = n**2 + 300

# Criando o gráfico
plt.figure(figsize=(10, 6))
plt.plot(n, f_n, label='A1: f(n) = 2n^2 + 9n', color='blue')
plt.plot(n, g_n, label='A2: g(n) = 9n^2 + 15n', color='green')
plt.plot(n, h_n, label='A3: h(n) = n^2 + 300', color='red')

# Configurações do gráfico
plt.title('Análise Comportamental dos Algoritmos A1, A2 e A3')
plt.xlabel('Tamanho de Entrada (n)')
plt.ylabel('Função de Complexidade')
plt.legend()
plt.grid(True)

# Exibindo o gráfico
plt.show()
```

Com esse código, o gráfico plotado será:



Ex04) Matriz de Strassen (meu Deus Clerivaldo q código gigante)

Matriz Quadradas de Strassen

```
#include <stdio.h>
#include <stdlib.h>

// função para alocar memória para uma matriz
int** alocaMatriz(int n) {
    int** matriz = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        matriz[i] = (int*)malloc(n * sizeof(int));
    }
    return matriz;
}

// função para liberar memória de uma matriz
void liberaMatriz(int** matriz, int n) {
    for (int i = 0; i < n; i++) {
        free(matriz[i]);
    }
    free(matriz);
}

// Função para somar duas matrizes
void somaMatrizes(int n, int** A, int** B, int** C) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}

// Função para subtrair duas matrizes
void subtraiMatrizes(int n, int** A, int** B, int** C) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
}

// Função de multiplicação de matrizes usando o método de Strassen (gigantesco meu
Deus Clerivaldo)
void strassen(int n, int** A, int** B, int** C) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
    } else {
        int k = n / 2;
        int** A11 = alocaMatriz(k);
        int** A12 = alocaMatriz(k);
        int** A21 = alocaMatriz(k);
        int** A22 = alocaMatriz(k);
        int** B11 = alocaMatriz(k);
        int** B12 = alocaMatriz(k);
```

```
        int** B21 = alocaMatriz(k);
        int** B22 = alocaMatriz(k);
        int** C11 = alocaMatriz(k);
        int** C12 = alocaMatriz(k);
        int** C21 = alocaMatriz(k);
        int** C22 = alocaMatriz(k);
        int** M1 = alocaMatriz(k);
        int** M2 = alocaMatriz(k);
        int** M3 = alocaMatriz(k);
        int** M4 = alocaMatriz(k);
        int** M5 = alocaMatriz(k);
        int** M6 = alocaMatriz(k);
        int** M7 = alocaMatriz(k);
        int** AResult = alocaMatriz(k);
        int** BResult = alocaMatriz(k);

        // dividindo matrizes em submatrizes
        for (int i = 0; i < k; i++) {
            for (int j = 0; j < k; j++) {
                A11[i][j] = A[i][j];
                A12[i][j] = A[i][j + k];
                A21[i][j] = A[i + k][j];
                A22[i][j] = A[i + k][j + k];

                B11[i][j] = B[i][j];
                B12[i][j] = B[i][j + k];
                B21[i][j] = B[i + k][j];
                B22[i][j] = B[i + k][j + k];
            }
        }

        // M1 = (A11 + A22) * (B11 + B22)
        somaMatrizes(k, A11, A22, AResult);
        somaMatrizes(k, B11, B22, BResult);
        strassen(k, AResult, BResult, M1);

        // M2 = (A21 + A22) * B11
        somaMatrizes(k, A21, A22, AResult);
        strassen(k, AResult, B11, M2);

        // M3 = A11 * (B12 - B22)
        subtraiMatrizes(k, B12, B22, BResult);
        strassen(k, A11, BResult, M3);

        // M4 = A22 * (B21 - B11)
        subtraiMatrizes(k, B21, B11, BResult);
        strassen(k, A22, BResult, M4);

        // M5 = (A11 + A12) * B22
        somaMatrizes(k, A11, A12, AResult);
        strassen(k, AResult, B22, M5);

        // M6 = (A21 - A11) * (B11 + B12)
        subtraiMatrizes(k, A21, A11, AResult);
        somaMatrizes(k, B11, B12, BResult);
```

```
        strassen(k, AResult, BResult, M6);

        // M7 = (A12 - A22) * (B21 + B22)
        subtraiMatrizes(k, A12, A22, AResult);
        somaMatrizes(k, B21, B22, BResult);
        strassen(k, AResult, BResult, M7);

        // C11 = M1 + M4 - M5 + M7
        somaMatrizes(k, M1, M4, AResult);
        subtraiMatrizes(k, AResult, M5, BResult);
        somaMatrizes(k, BResult, M7, C11);

        // C12 = M3 + M5
        somaMatrizes(k, M3, M5, C12);

        // C21 = M2 + M4
        somaMatrizes(k, M2, M4, C21);

        // C22 = M1 - M2 + M3 + M6
        subtraiMatrizes(k, M1, M2, AResult);
        somaMatrizes(k, AResult, M3, BResult);
        somaMatrizes(k, BResult, M6, C22);

        // Combinando submatrizes em uma matriz completa
        for (int i = 0; i < k; i++) {
            for (int j = 0; j < k; j++) {
                C[i][j] = C11[i][j];
                C[i][j + k] = C12[i][j];
                C[i + k][j] = C21[i][j];
                C[i + k][j + k] = C22[i][j];
            }
        }

        // Liberando a memória
        liberaMatriz(A11, k);
        liberaMatriz(A12, k);
        liberaMatriz(A21, k);
        liberaMatriz(A22, k);
        liberaMatriz(B11, k);
        liberaMatriz(B12, k);
        liberaMatriz(B21, k);
        liberaMatriz(B22, k);
        liberaMatriz(C11, k);
        liberaMatriz(C12, k);
        liberaMatriz(C21, k);
        liberaMatriz(C22, k);
        liberaMatriz(M1, k);
        liberaMatriz(M2, k);
        liberaMatriz(M3, k);
        liberaMatriz(M4, k);
        liberaMatriz(M5, k);
        liberaMatriz(M6, k);
        liberaMatriz(M7, k);
        liberaMatriz(AResult, k);
        liberaMatriz(BResult, k);
```

```
    }
}

int main() {
    int n = 4; // Dimensão da matriz (deve ser uma potência de 2 para simplificação)
    int** A = alocaMatriz(n);
    int** B = alocaMatriz(n);
    int** C = alocaMatriz(n);

    // Inicializando as matrizes A e B
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = i + j;
            B[i][j] = i - j;
        }
    }

    // Executando a multiplicação de Strassen
    strassen(n, A, B, C);

    // Exibindo a matriz resultante C
    printf("Matriz Resultante C:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    // Liberando memória das matrizes
    liberaMatriz(A, n);
    liberaMatriz(B, n);
    liberaMatriz(C, n);

    return 0;
}
```