Medindo a Complexidade do Código

Mário Leite

...

Escrever um programa é relativamente simples; o desafio está em medir sua **complexidade** - que aqui não significa "dificuldade" - mas sim, a relação com o tempo e recursos de execução. Complexidade de código é uma medida de quão complicado ou difícil de entender/manter um programa, e pode ser vista de duas formas principais:

1. Complexidade de tempo (*Time Complexity*)
    - Refere-se a quanto tempo o programa leva para rodar em função do tamanho da entrada. Por exemplo: um algoritmo que testa primos verificando todos os números até n tem complexidade **O(n²)**; um algoritmo do tipo **6k ± 1** tem **O(√n)**.
2. Complexidade de manutenção (ou ciclomática)
    - Mede quantos caminhos diferentes existem no código, como decisões condicionais (***if***, ***while***, ***for***).
    - Quanto maior, mais difícil testar, entender e dar manutenção no código.

Para medir complexidade podemos considerar fazê-lo de duas maneiras:

1. Contar quantas operações o algoritmo executa em função do tamanho da entrada, com quatro tipos de notações:

- Notações comuns:
    - **O(1)** constante
    - **O(n)** linear ==> *complexidade aumenta linearmente com tamanho da entrada.*
    - **O(n²)** quadrática
    - **O(log n)** logarítmica

2. Complexidade ciclomática

- Calcular quantos caminhos independentes existem no código:
    - Fórmula de" McCabe": **M = E − N + 2P**, onde:
        - E = número de arestas no grafo do fluxo do programa
        - N = número de nós
        - P = número de componentes conectados

Exemplo: cada ***if***, ***while*** ou ***for*** aumenta a complexidade.

Medir a complexidade não é apenas teoria: é o que diferencia um código que roda em segundos de outro que pode travar por horas. Deste modo, vamos apresentar um programa para gerar os seis primeiros números perfeitos, codificado em **C**, porém de duas formas diferentes, para mostrar as complexidades das duas versões. Na primeira versão "ingênua" (com força bruta) para cada número **n** o teste percorre **n/2 divisores → O(n)**; Já na segunda versão usando" Números de Mersenne" (mais eficiente), em vez de testar todos os números até encontrar um perfeito é usado o algoritmo do tipo "*Mersenne Primes",* tornando o código muito mais eficiente. Então, como é preciso encontrar até o quarto número perfeito, a primeira versão é lenta demais, com **Complexidade: O(n²)** se considerar a busca acumulada. Compare o tempo de processamento do programa **"NumerosPerfeitos1.C"** (com força bruta - **figura 1a**) com o do programa **"NumerosPerfeitos2.C"** (método de Mersenne – **figura 1b**). A complexidade da segunda versão é bem menor; com mais detalhes e exibindo **6** números, em vez de apenas **4**. Para explicar melhor a Complexidade um outro código, em Phyton, é apresentado para medir as complexidades dos dois programas, exibindo dois tipos de gráficos (**figuras 2a e 2b**), mostrando que o método da Força Bruta é mais complexo que o de Mersenne.

```c
// NumerosPerfeitos1.C (usando Força Bruta)
#include <stdio.h>
#include <time.h>

int ehPerfeito(long long n) {
    long long soma = 0;
    for (long long i = 1; i <= n / 2; i++) {
        if (n % i == 0) soma += i;
    }
    return soma == n;
}

int main() {
    clock_t inicio, fim;
    double tempo;
    int count = 0;
    long long num = 2;

    printf("Versao Forca Bruta:\n");
    inicio = clock();  // início da medição

    while (count < 6) {
        if (ehPerfeito(num)) {
            printf("%lld\n", num);
            count++;
        }
        num++;
    }
    fim = clock();  // fim da medição
    tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;
    printf("Tempo gasto: %.6f segundos\n", tempo);
    return 0;
```

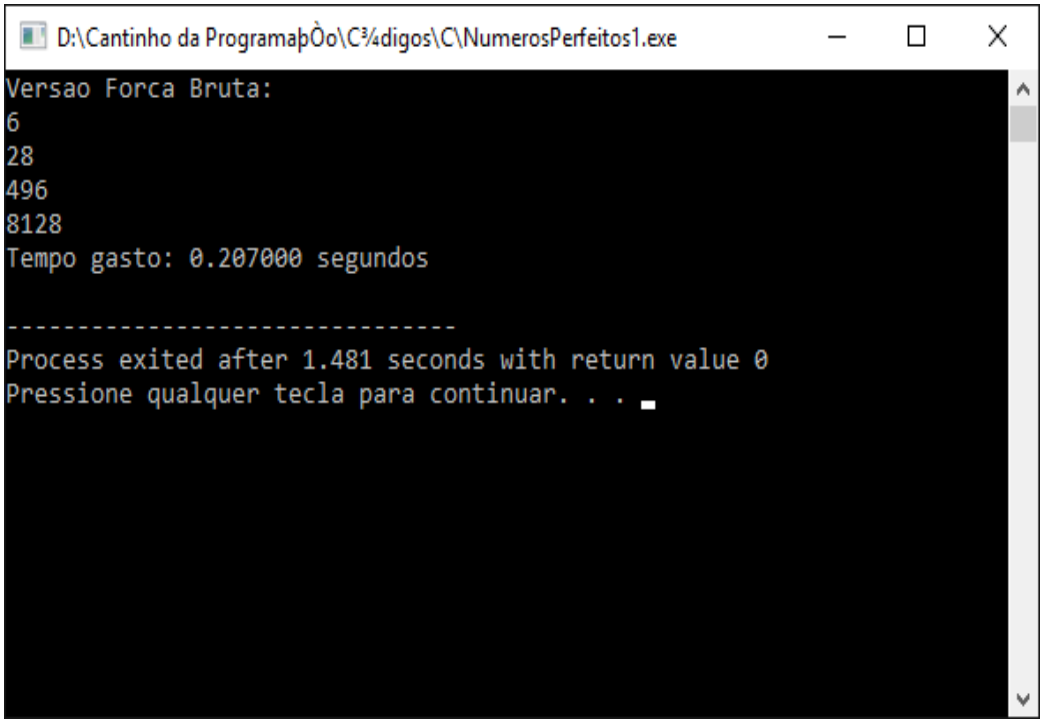

Figura 1a - Saída do programa "NumerosPerfeitos1.C"

```c
#NumerosPerfeitos2.C (usando Números de Mersenne)
#include <stdio.h>
#include <time.h>

int ehPrimo(long long n) {
    if (n < 2) return 0;
    for (long long i = 2; i * i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

int main() {
    clock_t inicio, fim;
    double tempo;
    int count = 0;

    printf("\nVersao Mersenne:\n");
    inicio = clock();  // início da medição

    for (int p = 2; count < 6; p++) {
        long long mersenne = (1LL << p) - 1;  // 2^p - 1
        if (ehPrimo(mersenne)) {
            long long perfeito = (1LL << (p - 1)) * mersenne;
            printf("%lld\n", perfeito);
            count++;
        }
    }
    fim = clock();  // fim da medição
    tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;

    printf("Tempo gasto: %.6f segundos\n", tempo);
    return 0;
}
```

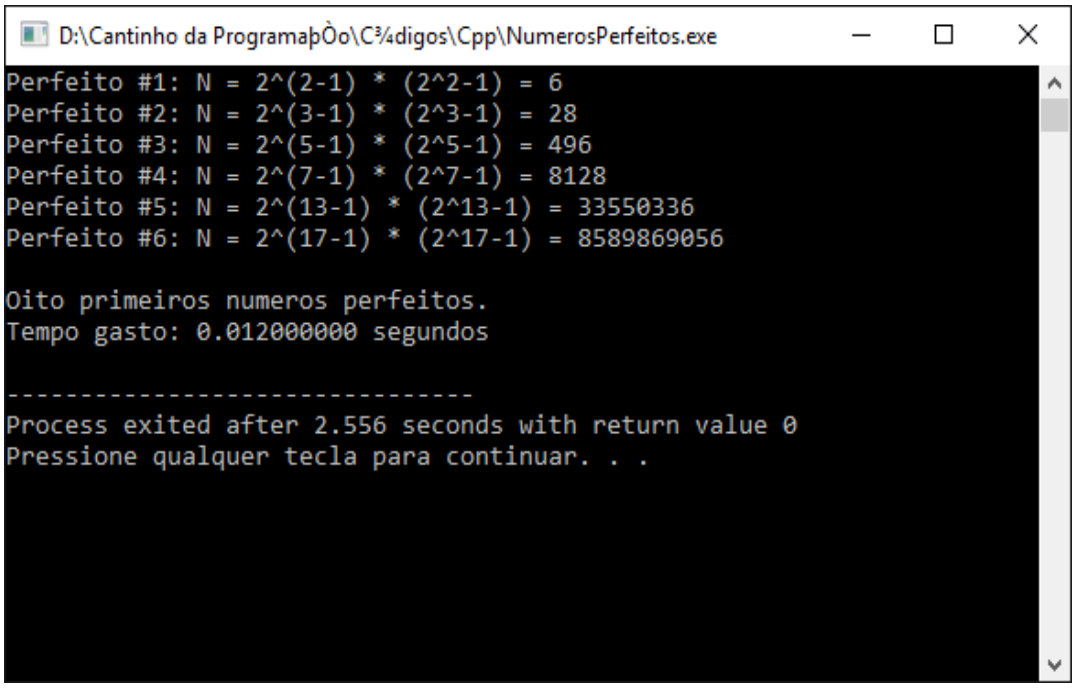

Figura 1b - Saída do programa “NumerosPerfeitos2.C”

```
'''
ExibeComplexidade.py
Calcula e exibe gráficos mostrando a Complexidade de dois códigos diferentes.
-------------------------------------------------------------------------------
'''
import time
import matplotlib.pyplot as plt

# ---------- Algoritmos ----------
# Método de "Força Bruta" para verificar se o número é perfeito
def VerficarPerfeitoBruta(n):
    soma = 0
    for i in range(1, n // 2 + 1):
        if(n % i == 0):
            soma += i
    return soma == n
#-------------------------------------------------------------------------------
def EncontrarPerfeitosBruta(qtd):
    LstEncontrados = []  #lista de números perfeitos encontrados
    n = 2
    while len(LstEncontrados) < qtd:
        if(VerficarPerfeitoBruta(n)):
            LstEncontrados.append(n)
        n += 1
    return LstEncontrados
#-------------------------------------------------------------------------------
# Método de "Mersenne" para verificar se o número é perfeito
def VerificarPrimo(n):
    if(n < 2):
        return False
    for i in range(2, int(n**0.5) + 1):
        if(n % i == 0):
            return False
    return True
#-------------------------------------------------------------------------------
def EncontrarPerfeitosMersenne(qtd):
    LstEncontrados = []  #lista dos números perfeitos encontrados
    p = 2
    while(len(LstEncontrados) < qtd):
        mersenne = (1 << p) - 1  # 2^p - 1
        if(VerificarPrimo(mersenne)):
            perfeito = (1 << (p - 1)) * mersenne
            LstEncontrados.append(perfeito)
        p += 1
    return LstEncontrados
# ---------- Comparação de tempo ----------
def MedirTempo(func, qtd):
    inicio = time.time()
    nums = func(qtd)
    fim = time.time()
    return fim - inicio, nums
#-------------------------------------------------------------------------------
def CompararAlgoritmos(qtd):
    LstTemposBruto = []   #lista dos tempos com Força Bruta
    LstTemposMersenne = [] #lista dos tempos com Método Mersenne
    for i in range(1, qtd + 1):
        tempo1, _ = MedirTempo(EncontrarPerfeitosBruta, i)
        tempo2, _ = MedirTempo(EncontrarPerfeitosMersenne, i)
```

```
        LstTemposBruto.append(tempo1)
        LstTemposMersenne.append(tempo2)

    x = range(1, qtd + 1)

    # ---------- Gráfico de Linha ----------
    plt.figure(figsize=(10,6))
    plt.plot(x, LstTemposBruto, 'r-o', label="Força Bruta")
    plt.plot(x, LstTemposMersenne, 'g-o', label="Mersenne")
    plt.xlabel("Quantidade de Números Perfeitos Calculados")
    plt.ylabel("Tempo (segundos)")
    plt.title("Comparação de Complexidade - Gráfico de Linhas")
    plt.legend()
    plt.grid(True)
    plt.show()

    # ---------- Gráfico de Área ----------
    plt.figure(figsize=(10,6))
    plt.fill_between(x,LstTemposBruto,color="red",alpha=0.5,label="Força Bruta")
    plt.fill_between(x,LstTemposMersenne,color="green",alpha=0.5,label="Mersenne")
    plt.xlabel("Quantidade de Números Perfeitos Calculados")
    plt.ylabel("Tempo (segundos)")
    plt.title("Comparação de Complexidade - Gráfico de Área")
    plt.legend(loc="upper left")
    plt.grid(True)
    plt.show()

#=============================================================================================
#Programa principal
if(__name__ == "__main__"):
    CompararAlgoritmos(4)  #acima de 4 o tempo com Força Bruta é extremamente alto
#Fim do programa " ExibeComplexidade"
```

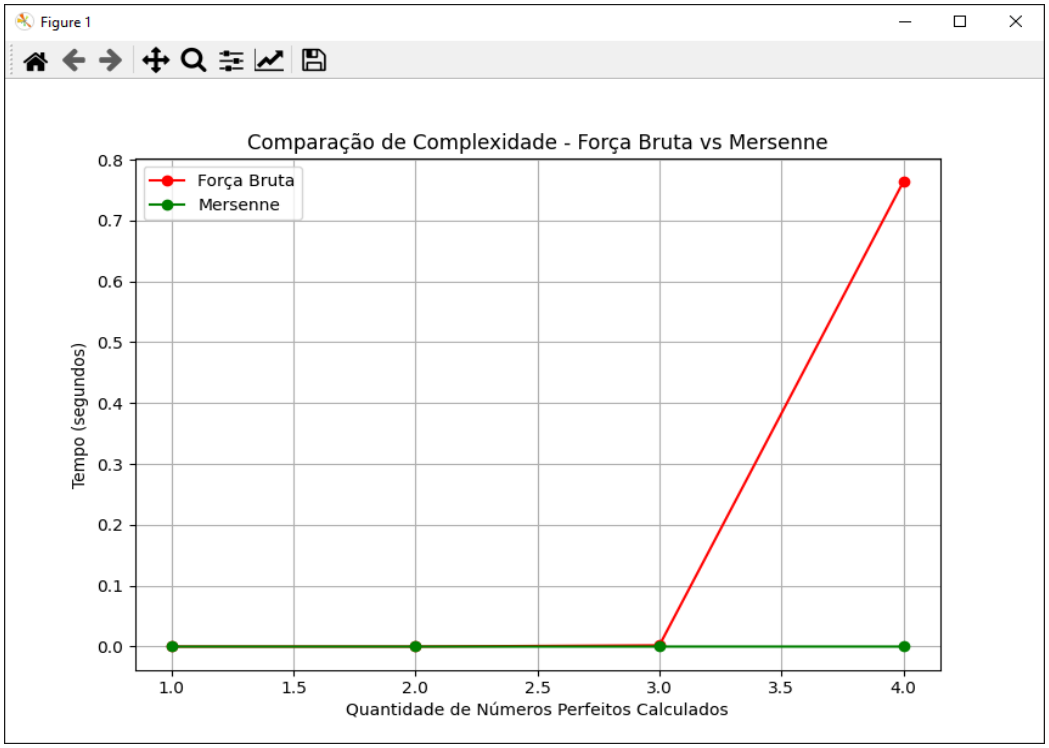

Figura 2a - Saída do programa “ExibeComplexidade”: gráficos de linhas

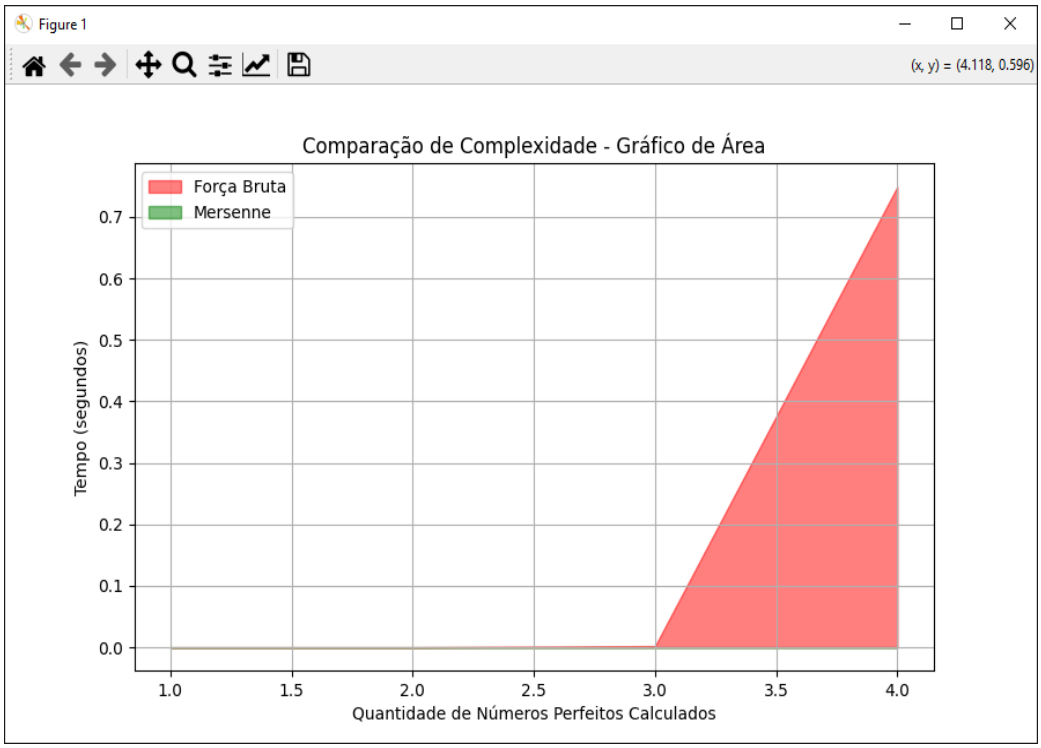

Figura 2b - Saída do programa “ExibeComplexidade”: gráficos de áreas