Recursividade - Estrutura de Dados

Gabriel Gaspar

17 novembro 2024

1 Introdução

Nesse trabalho, foram modificados programas em C++ já prontos, de geração randômica de números, de cálculo do máximo divisor comum de 200 valores, utilizando 3 algoritmos diferentes, com os números vindo do programa de geração randômica e um programa que faz o cálculo para achar valores na sequência de Fibonacci, tanto de maneira recursiva como iterativa.

2 Ambiente de desenvolvimento

Os programas foram desenvolvidos/modificados e testados em um computador "Samsung Galaxy Book 4 Pro", utilizando o Windows Subsystem for Linux (WSL), na edição Linux 5.15.153.1-microsoft-standard-WSL2, usando como sistema operacional convidado o Ubuntu 22.04.5 LTS, com as seguintes especificações:

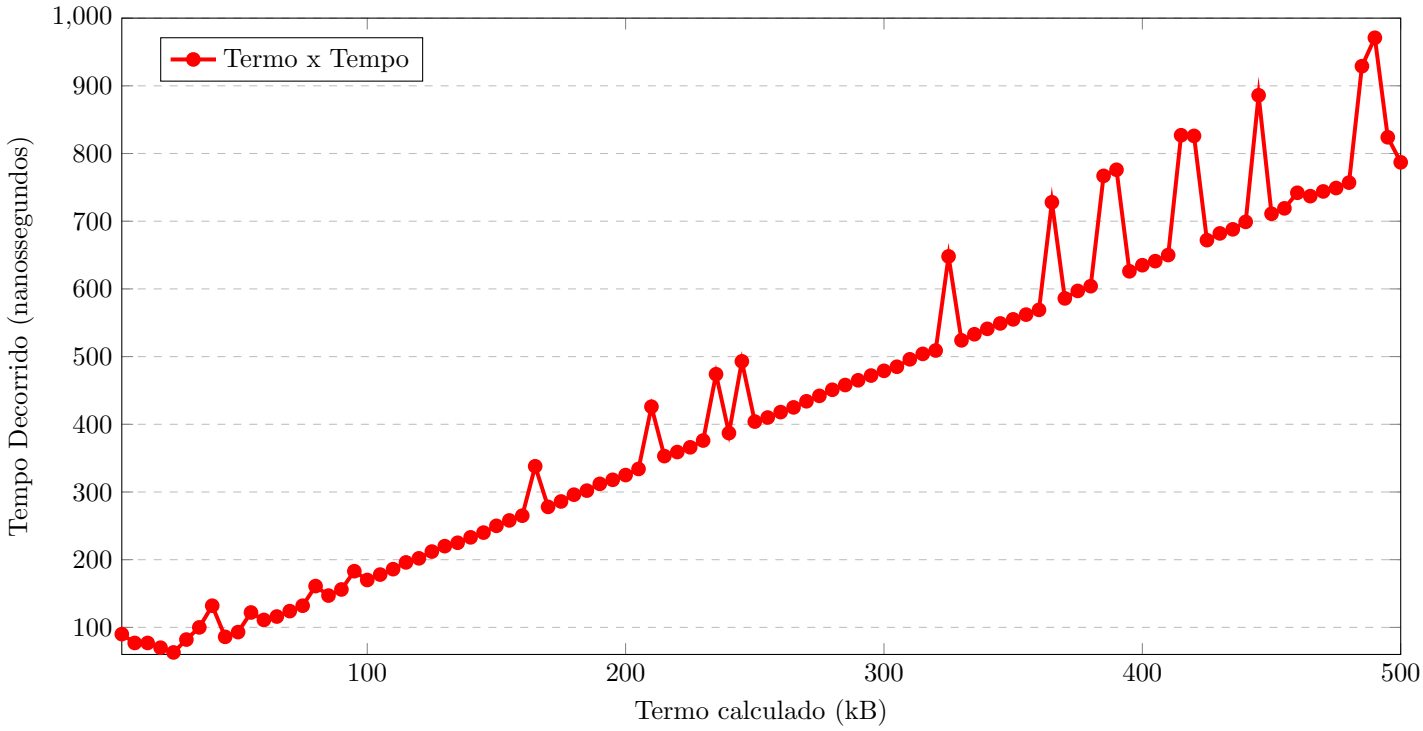
- Sistema Operacional: Windows 11 / Ubuntu 22.04.5 LTS
- Processador: Intel Core Ultra 7 155H
- Memória: 16GB DDR5 7467MT/s
- Armazenamento: SAMSUNG MZVL4512HBLU-00B
- IDE: Visual Studio Code 1.95.3

3 Fibonacci

Para calcular o Fibonacci, foram feitos dois programas, um usando iteratividade e outro usando recursividade, sendo que os dois começam no quinto termo e vão até alcançar o quingentésimo termo, crescendo em uma proporção de 5 em 5 por repetição. É possível notar que, no caso do cálculo dos termos de Fibonacci, o programa iterativo roda muito mais rápido quando comparado ao programa recursivo.

No programa iterativo, pode se perceber que a medição usando chrono e a medição usando getrusage tem diferenças. Possivelmente isso acontece porque o getrusage pega o tempo total em que o programa esteve rodando, enquanto o chrono, pega o tempo desde que se iniciou a contagem. Também é possível perceber que o tempo de sistema sempre fica nulo, demonstrando que o sistema só utiliza o modo de usuário para rodar o programa. O gráfico a seguir mostra o termo no qual o programa estava calculando pelo tempo demorado, em nanossegundos, utilizando o relógio de alta resolução. Foi utilizado nanossegundos, e não microssegundos como é pedido no Classroom, devido aos números em microssegundos estarem variando entre 0 e 1. Não coloquei a quantidade de RAM utilizada pelo programa por ela não ter alterações consideráveis. É importante notar que o programa foi manipulado para imprimir as coordenadas necessárias para facilitar a construção do gráfico no LaTeX.

Gráfico do termo calculado pelo tempo decorrido (iterativo)



Abaixo está o código do Fibonacci iterativo:

```cpp
#include <iostream>
#include <iomanip>
#include <chrono>
#include <sys/resource.h>
using namespace std;

unsigned long long int fibonacci(unsigned long long int valor) {
   if (valor <= 2)
      return 1;
   unsigned long long int n1 = 1; // 1 Termo
   unsigned long long int n2 = 1; // 2 Termo
   for(unsigned long long int i=2; i < valor; ++i){
      unsigned long long int anterior = n1;
      n1 = n2;
      n2 = anterior + n1;
   }
  return n2;
}

int main() {

   struct rusage r_usage;
   getrusage(RUSAGE_SELF,&r_usage);
   cout << r_usage.ru_utime.tv_sec << ":" << r_usage.ru_stime.tv_sec << std::endl;
   cout << "Uso de memória atual: " << r_usage.ru_maxrss << "KB" << endl;

   auto startH = std::chrono::high_resolution_clock::now();

    for (int i = 5; i <= 500; i += 5) {
        auto start = chrono::high_resolution_clock::now();
        fibonacci(i);
        auto stop = chrono::high_resolution_clock::now();
        auto d12 = chrono::duration_cast<std::chrono::nanoseconds>(stop - start);

        cout << i << ";" << d12.count() << endl;
    }

   auto stopH = std::chrono::high_resolution_clock::now();

   auto d1 = std::chrono::duration_cast<std::chrono::microseconds>(stopH - startH);
```

```
41
42    getrusage(RUSAGE_SELF, &r_usage);
43    auto userTime = r_usage.ru_utime.tv_sec * 1000000.0 + r_usage.ru_utime.tv_usec;
44    auto sysTime = r_usage.ru_stime.tv_sec * 1000000.0 + r_usage.ru_stime.tv_usec;
45
46    cout << endl;
47    cout << "O programa demorou " << setprecision(5) << d1.count() << " microssegundos para terminar de
       rodar." << endl;
48    cout << "O programa teve " << userTime << " microssegundos de tempo de usuário e " << sysTime << "
       microssegundos de tempo de sistema." << endl;
49    cout << "O programa está utilizando atualmente " << r_usage.ru_maxrss << " KB de memória RAM" <<
       endl;
50 }
```

Em relação ao Fibonacci de modo recursivo, é importante notar que o programa não termina de acordo com as instruções passadas na atividade, e o tempo para terminar conforme se avança de termo, é exponencial. O programa simplesmente roda por tempo indeterminado. Não foi feito nenhum gráfico pelo fato do programa nunca terminar e não encontrar do quinquagésimo quinto termo em diante, sendo o último termo encontrado, o quinquagésimo termo. Abaixo está o código do programa utilizado.

```
1  #include <iostream>
2  #include <iomanip>
3  #include <chrono>
4  #include <sys/resource.h>
5  using namespace std;
6
7  unsigned long long int fibonacci(unsigned long long int valor) {
8     if (valor <= 2)
9        return 1;
10    unsigned long long int n1 = 1; // 1 Termo
11    unsigned long long int n2 = 1; // 2 Termo
12    for(unsigned long long int i=2; i < valor; ++i){
13       unsigned long long int anterior = n1;
14       n1 = n2;
15       n2 = anterior + n1;
16    }
17   return n2;
18 }
19
20 int main() {
21
22    struct rusage r_usage;
23    getrusage(RUSAGE_SELF,&r_usage);
24    cout << r_usage.ru_utime.tv_sec << ":" << r_usage.ru_stime.tv_sec << std::endl;
25    cout << "Uso de memória atual: " << r_usage.ru_maxrss << "KB" << endl;
26
27    auto startH = chrono::high_resolution_clock::now();
28
29    for (int i = 5; i <= 500; i += 5) {
30       cout << i << ": " << fibonacci(i) << endl;
31    }
32
33    auto stopH = chrono::high_resolution_clock::now();
34
35    auto d1 = chrono::duration_cast<chrono::microseconds>(stopH - startH);
36
37    getrusage(RUSAGE_SELF, &r_usage);
38    auto userTime = r_usage.ru_utime.tv_sec * 1000000 + r_usage.ru_utime.tv_usec;
39    auto sysTime = r_usage.ru_stime.tv_sec * 1000000 + r_usage.ru_stime.tv_usec;
40
41    cout << endl;
42    cout << "O programa demorou " << setprecision(5) << d1.count() << " microssegundos para terminar de
       rodar." << endl;
43    cout << "O programa teve " << userTime << " microssegundos de tempo de usuário e " << sysTime << "
       microssegundos de tempo de sistema." << endl;
44 }
```

4 Máximo Divisor Comum

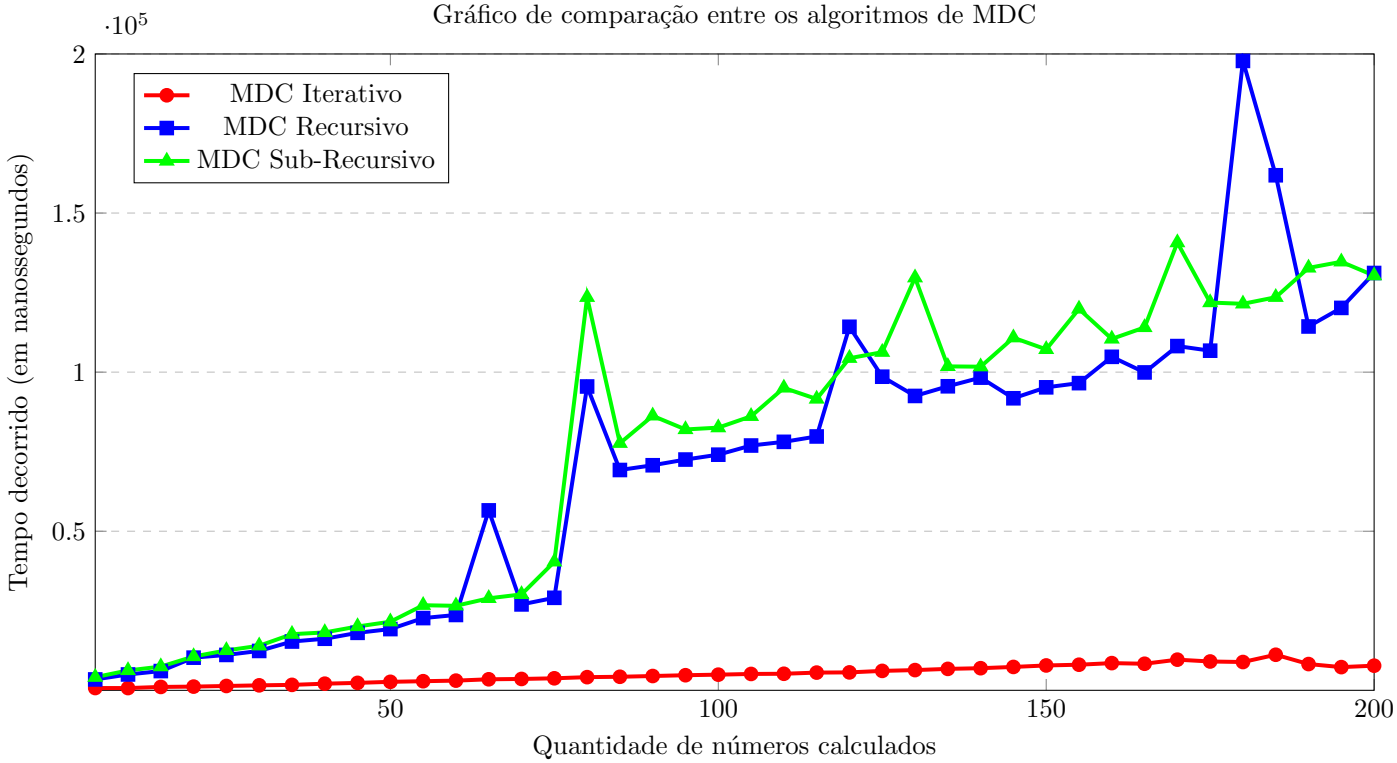
Ao rodar o código do programa gera_numeros_mdc.cpp, que está no Classroom, com o seguinte parâmetro:

```
./"gera_numeros_mdc" > teste.csv
```

foi possível passar os números gerados e que foram impressos para um arquivo CSV, prontos para a leitura do programa que faz o cálculo do máximo divisor comum com os 3 algoritmos diferentes, sendo eles iteratividade, recursividade e por fim, sub recursividade. Foi separado cada algoritmo em um programa diferente, para facilitar a montagem dos gráficos em LaTeX.

No gráfico a seguir, a unidade de tempo utilizada foi o nanossegundo, já que ao usar microssegundos no MDC iterativo os valores ficam muito baixos, então para manter um padrão, todos os programas usam a mesma unidade de tempo.

Gráfico de comparação entre os algoritmos de MDC



5 Códigos do Máximo Divisor Comum

O programa a seguir faz os cálculos do Máximo Divisor Comum utilizando os 3 algoritmos. Para fazer o gráfico anterior, o programa foi manipulado para mostrar somente o tempo de cada algoritmo.

```
#include <chrono>

#include <iomanip>
#include <sys/resource.h>
#include <unistd.h>

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

struct Registro {
    int n1;
    int n2;
};

int mdc_recursivo(int m, int n)
{
    if( m == n)
    {
        return m;
    }
    if( m-n >= n)
    {
        return mdc_recursivo(m-n,n);
```

```
    }
    return mdc_recursivo(n,m-n);
}

int mdc_iterativo(int m, int n)
{
    if( n == 0)
    {
        return m;
    }
    while( m % n != 0)
    {
        int resto = m % n;
        m = n;
        n = resto;
    }
    return n;

}

int mdc_sub_recursivo(int m, int n)
{
    if( m == n)
    {
        return m;
    }
    if( m-n >= n)
    {
        return mdc_sub_recursivo(m-n,n);
    }
    return mdc_sub_recursivo(n,m-n);
}

std::vector<Registro> lerArquivoCSV(const std::string& nomeArquivo) {
    std::vector<Registro> registros;
    std::ifstream arquivo(nomeArquivo);

    if (!arquivo.is_open()) {
        std::cerr << "Erro ao abrir o arquivo " << nomeArquivo << std::endl;
        return registros;
    }

    std::string linha;
    while (std::getline(arquivo, linha)) {
        std::size_t pos = linha.find(";");
        if (pos != std::string::npos) {
            Registro registro;
            registro.n1 = atoi(linha.substr(0, pos).c_str());
            registro.n2 = atoi(linha.substr(pos + 1).c_str());
            //garantindo que o maior está em n1
            if( registro.n1 < registro.n2 ){
               int aux = registro.n2;
               registro.n2 = registro.n1;
               registro.n1 = aux;
            }
            registros.push_back(registro);
        }
    }

    arquivo.close();
    return registros;
}

int main() {
    std::string nomeArquivo = "teste.csv";
    std::vector<Registro> dados = lerArquivoCSV(nomeArquivo);

    for( int i=5; i<=200; i+=5){

       //iterativo
       auto start_mdciterativo = std::chrono::steady_clock::now();
       for( int k=0;k<i;++k){
           mdc_iterativo(dados[k].n1, dados[k].n2);
       }
```

```
        auto stop_mdciterativo = std::chrono::steady_clock::now();

        //recursivo
        auto start_mdcrecursivo = std::chrono::steady_clock::now();
        for( int k=0;k<i;++k){
            mdc_recursivo(dados[k].n1, dados[k].n2);
        }
        auto stop_mdcrecursivo = std::chrono::steady_clock::now();

        //recursivo sub
        auto start_mdcsubrecursivo = std::chrono::steady_clock::now();
        for( int k=0;k<i;++k){
            mdc_sub_recursivo(dados[k].n1, dados[k].n2);
        }
        auto stop_mdcsubrecursivo = std::chrono::steady_clock::now();

        auto d1 = std::chrono::duration_cast<std::chrono::nanoseconds>(stop_mdciterativo -
    start_mdciterativo);
        auto d2 = std::chrono::duration_cast<std::chrono::nanoseconds>(stop_mdcrecursivo -
    start_mdcrecursivo);
        auto d3 = std::chrono::duration_cast<std::chrono::nanoseconds>(stop_mdcsubrecursivo -
    start_mdcsubrecursivo);

        std::cout << i << ";" << d1.count() << ";" << d2.count() << ";" << d3.count() << ";" << std::
    endl;

        /*
         cout pro iterativo: std::cout << "(" << i << ", " << d1.count() << ") ";
         cout pro recursivo: std::cout << "(" << i << ", " << d2.count() << ") ";
         cout pro subrecursivo: std::cout << "(" << i << ", " << d3.count() << ") ";
        */
    }

    return 0;
}
```

6 CMake

A parte do CMake foi um tanto complicada de se entender. No Classroom, está anexado um arquivo "CMake.zip", com os arquivos necessários para o CMake funcionar, como o "CMakeLists.txt"e o "Makefile". Para compilar corretamente os códigos, é necessário entrar na pasta "build"e rodar o seguinte comando, dentro de uma CLI:

```
cmake --build .
```

7 Conclusão

Finalizando, com esse trabalho foi possível perceber que a recursividade é um artefato muito útil na vida de um programador, deixando o código mais enxuto e mais fácil de se trabalhar com. Mas não é sempre assim, como nota-se no programa que calcula Fibonacci usando recursividade. Neste programa, quando chega no quinquagésimo termo, o programa começa a demorar muito, diferentemente da versão iterativa do mesmo programa, no qual consegue alcançar o quingentésimo termo em tempos menores que 0.0001 segundos.

No programa que calcula o Máximo Divisor Comum acontece a mesma coisa, a versão iterativa do programa performa melhor que as versões com os outros dois algoritmos, sendo que o iterativo consegue alcançar tempos 10 vezes mais rápidos que o recursivo e o sub recursivo.