Comparação empírica sobre tempo de execução baseados em diferentes paradigmas

Carlos Eduardo da Silva
Universidade Federal de São João del–Rei
carlos.aeduardo@aluno.ufsj.edu.br

Abstract

Um algoritmo pode ser descrito em varias formas diferentes, diante está variação na implementação de um algoritmo podemos ter várias possibilidades, o objetivo deste trabalho é comparar o tempo de execução da sequência de fibonacci para diferentes algoritmos e linguagens de programação com paradigmas diferentes.

1 Introdução

Trabalho proposto na disciplina de 'Conceitos de linguagens de programação'. Tem como objetivo desenvolver soluções para o cálculo do enésimo termo da sequência de fibonacci utilizando três linguagens de paradigmas de programação distintas e comparar o tempo de execução em cada linguagem.

1.1 Sequência de Fibonacci

É a sequência numérica proposta pelo matemático Leonardo Pisa, mais conhecido como Fibonacci. Foi a partir de um problema criado por ele que o mesmo detectou a existência de uma regularidade matemática.

Trata-se do exemplo clássico dos coelhos, em que Fibonacci descreve o crescimento de uma população desses animais.

A sequência é definida pela fórmula na figura 1 abaixo:

$$F(n) = \begin{cases} 0, \\ 1, \\ F(n-1) + F(n-2) \end{cases}$$

Figure 1: Fórmula fibonacci

Assim, começando pelo 1, essa sequência é formada somando cada numeral com o numeral que o antecede. No caso do 1, repete-se esse numeral e soma-se, ou seja, 1 + 1 = 2.

De seguida soma-se o resultado com o numeral que o antecede, ou seja, 2 + 1 = 3 e assim sucessivamente, como mostra a figura 2.

É importante destacar que a sequência de Fibonacci é infinita. Portanto, o ideal é que você defina um ou mais valores que tenha como objetivo.

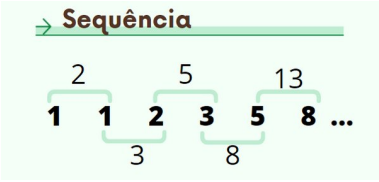

Figure 2: Sequencia sucessiva

2 Materiais e Métodos

2.1 Paradigmas

Para a realização deste projeto desenvolvemos em três linguagens de paradigmas de programação diferentes(Orientado a objetos, funcional e imperativo) e três algoritmos para cada linguagem.

Imperativo

O fundamento da programação imperativa é o conceito de Máquina de Turing, que nada mais é que uma abstração matemática que corresponde ao conjunto de funções computáveis.

A Máquina de Turing foi aprimorada por John Von Neumann a uma arquitetura de computadores que fundamenta os computadores construídos até hoje mostrado na figura 3.

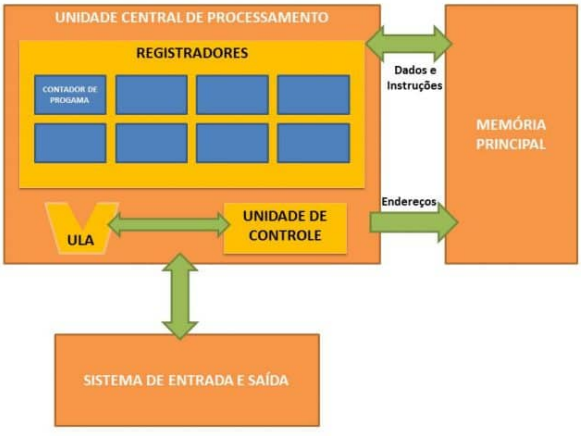

Figure 3: Arquitetura de Von Neumann

Nesta arquitetura, tanto os dados como os programas são armazenados na mesma memória, e a CPU que processa as instruções, é separada da memória. Dessa forma, os dados e instruções devem ser transportados da memória para a CPU e os resultados das operações realizadas na CPU devem ser devolvidos para a memória.

Orientação a Objetos

A programação orientada a objetos (OO) é um recurso usado em muitas das linguagens atuais.
Estrutura que simula uma entidade do mundo real, identificando características (atributos) e ações (métodos) próprias dessa entidade.
Pode-se dizer também que uma classe descreve um conjunto de indivíduos que compartilham características e ações em comum.
Os quatro pilares da orientação a objetos é mostrado na figura 4.

Figure 4: Pilares da POO

É uma construção que representa um tipo abstrato de dados:

- Possui um nome que representa o seu propósito;
- Define uma organização para armazenamento de dados;
- Define um conjunto de métodos (subprogramas) que permitem manipular suas instâncias;
- Oculta a sua a estrutura (complexidade) interna.

Programação Funcional

Considera o programa como uma função matemática (ou um conjunto delas).
Baseia-se no conceito matemático de função, em que para cada elemento do seu conjunto domínio (entrada) há apenas um elemento no seu conjunto contradomínio (saída) evitando estados ou dados mutáveis. Ela enfatiza a aplicação de funções, em contraste da programação imperativa, que enfatiza mudanças no estado do programa. Enfatizando as expressões ao invés de comandos, as expressões são utilizados para cálculo de valores com dados imutáveis.
Os programas seguem o formato do cálculo lambda. A programação funcional trata as funções de forma em que estas possam ser passadas como parâmetro e valores para outras e funções e podendo ter o resultado armazenado em uma constante.

As linguagens funcionais possuem mais algumas características importantes:

- Ausência de efeito colateral
- Independência da ordem de avaliação das expressões
- Transparência referencial
- Avaliação preguiçosa
- Recursão em cauda

2.2 Linguagens

C

A linguagem foi criada em 1972 e é uma linguagem imperativa e procedural para implementação de sistemas. Seus pontos de design foram para ele ser compilado, fornecendo acesso irrestrito à memória e baixos requerimentos do hardware.

Python

É uma linguagem de programação de alto nível, interpretada de script, criada em 1991 e de multiparadigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedural. Possui tipagem dinâmica e uma de suas principais características é permitir a fácil leitura do código e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens.

Haskell

Linguagem de paradigma funcional pura, criada em 1990 om tipagem estática polimórfica inferida e de avaliação preguiçosa, bem diferente da maioria das outras linguagens.
Ser uma linguagem funcional pura significa que todas as construções da linguagem giram em torno de funções que não possuem efeitos colaterais, e que todas as entradas estão definidas no tipo da função e que possuem apenas um valor como saída.

2.3 Estratégias e recursos

Foi utilizado como citado acima, foi utilizado três algoritmos distintos para cada linguagem (exceto para programação funcional que não dispõe do recurso de iteração, mais conhecido como laço ou loop).

Recursividade

Consiste na ideia de fragmentar o problema em vários problemas menores do mesmo tipo, e ao resolver os problemas menores (dividindo-os em problemas ainda menores, se necessário) e combinar as soluções dos problemas menores forma-se a solução final. De modo geral, uma definição de função recursiva é dividida em duas partes:

- Há um ou mais casos base que dizem o que fazer em situações simples, onde não é necessária nenhuma recursão.
  Nestes casos a resposta pode ser dada de imediato, sem chamar recursivamente a função sendo definida.
  Isso garante que a recursão eventualmente possa parar e não ficar 'preso' infinitamente como na figura 5.
- Há um ou mais casos recursivos que são mais gerais, e definem a função em termos de uma chamada mais simples a si mesma.

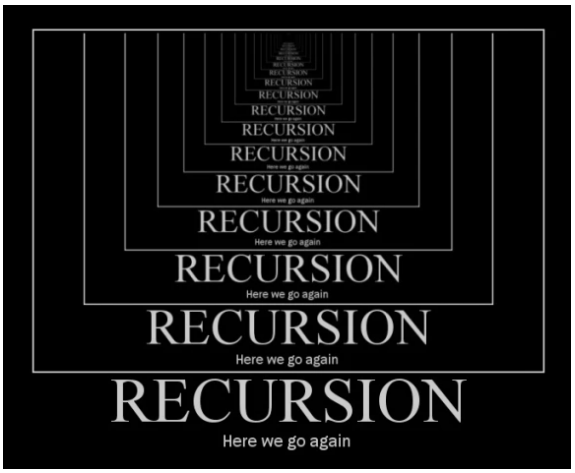

Figure 5: Recursividade sem caso base

Recursividade em cauda

Técnica de recursão que faz menos uso de memória durante o processo de empilhamento, o que a torna mais rápida que a recursão comum.
Um dos problemas não citados da recursividade comum é que a cada chamada recursiva realizada, é necessário guardar a posição do código onde foi feita a chamada para que continue a partir dali assim que receber o resultado.
Em uma recursão de cauda, não é necessário guardar a posição onde foi feita a chamada, visto que a chamada recursiva é a última operação realizada pela função.
Comparativo entre as duas técnicas de recursão em um exemplo na figura 6

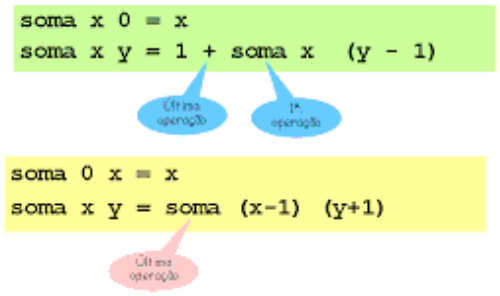

Figure 6: Recursão x Recursão em cauda

Dois pontos a serem destacados:

- Note que a primeira operação feita é a chamada recursiva. Assim, o programa precisa voltar no estado de todas as chamadas (logo, o programa precisa salvar este estado na memoria) para retornar o valor da função e em seguida fazer a ultima operação.
- Na técnica de recursão em cauda, a ultima operação da função é a chamada recursiva.

Laço de repetição

Estrutura simples de repetição condicional disponibilizado na maioria dos paradigmas de programação. A figura 11 exemplifica o fluxo.

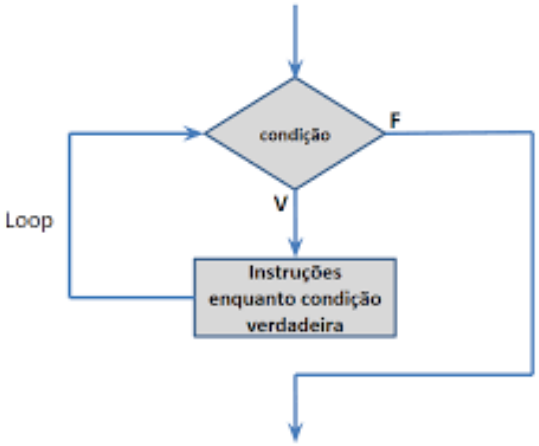

Figure 7: Fluxograma da estrutura de repetição

Enquanto a condição for verdadeira, será executado o bloco de código. No que permite fazer o somatório até enésimo termo da sequência que queremos calcular.

3 Resultados e Discussão

Os testes realizados para sequência do decimo (10º) até nongentésimo nonagésimo (990º) termo com um intervalo de 10 para cada termo. (exemplo: 10, 20, 30, 40...) Porém, para recursão comum foi restringido até quinquagésimo (50º) termo, e entenderemos isso nos teste.

3.1 Recursão

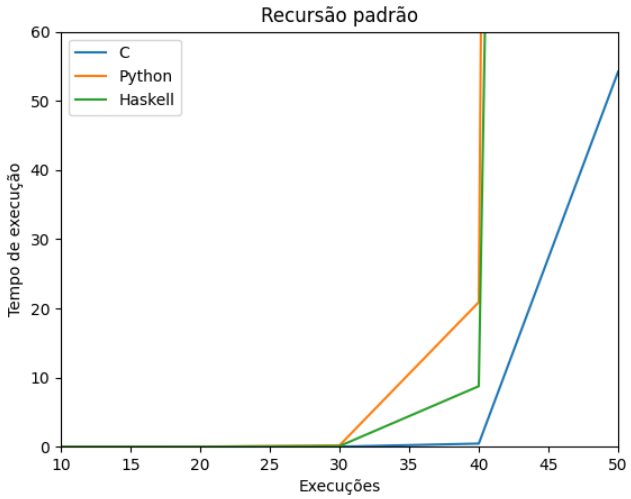

Figure 8: Tempo de execução recursivo

Como citado acima, os testes em recursão comum foi restringido a um intervalo bem menor, devido a principal desvantagem da recursão padrão. Os tempos para sequencias com o enésimo termo relativamente pequeno, necessitou de muito tempo. Analisando a figura 8 a execução da sequência até o trigésimo (30º) valor, está equiparado para as três linguagens, em breve analisaremos este intervalo mais de perto. Para valores maiores que trinta, conseguimos notar uma variação muito grande.
No quadragésimo (40º) valor, a linguagem orientada a objetos (python) tem o pior desempenho, demorando pouco mais do dobro de tempo da linguagem funcional e 20 vezes mais lenta que a linguagem imperativa.
Por fim, para o ultimo valor de teste dessa sequencia, temos uma crescimento grandezas proporcionais similares ao anterior, exceto pelo fato que a linguagem orientada a objetos (python) foi 40 vezes mais lenta que a linguagem imperativa. Agora vamos analisando o intervalo até o trigésimo (30º) valor mais de perto na figura 9.

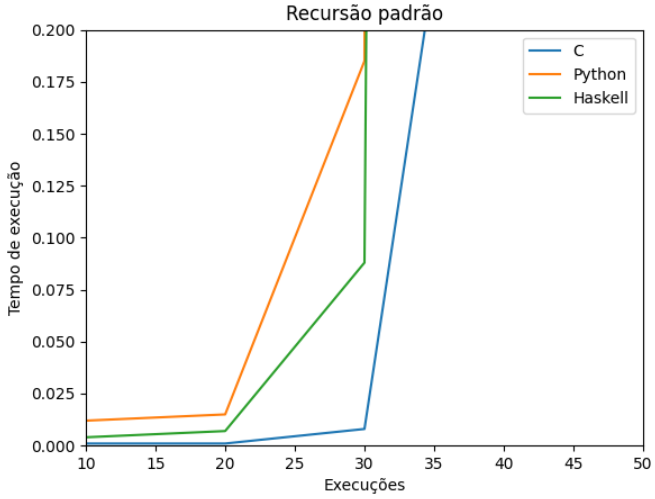

Figure 9: Tempo de execução recursivo

Com base neste grafico, podemos observar que é a linguagem orientada a objetos e funcional, obedecem uma grandeza proporcional de tempo gasto na execução (incluindo a linguagem imperativa até o quadragésimo valor).

3.2 Recursão em cauda

A estratégia da recursão em cauda, torna-se viavel os testes até o valor estipulado deviado a vantagem ja citada.
E podemos comprovar sua enficiencia como mostra a figura 10

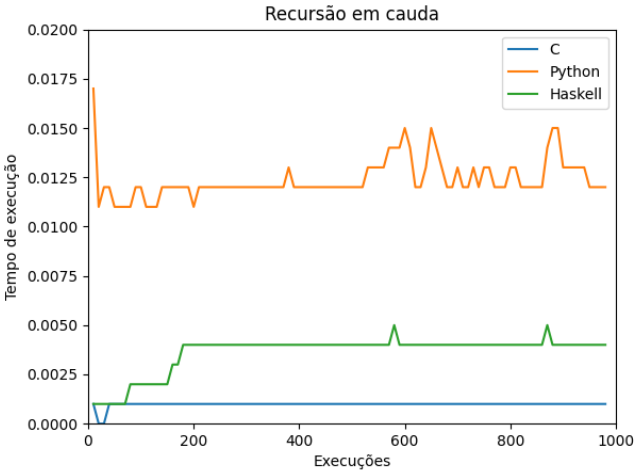

Figure 10: Tempo de execução recursivo em cauda

A recursão em cauda pode se considerar constante para cada linguagem distinta, apresentando uma variação insignificante nos teste.

3.3 Laço de repetição

Os testes realizados somente em dois paradigmas, ja que o paradigma funcional não provém de tal recurso.

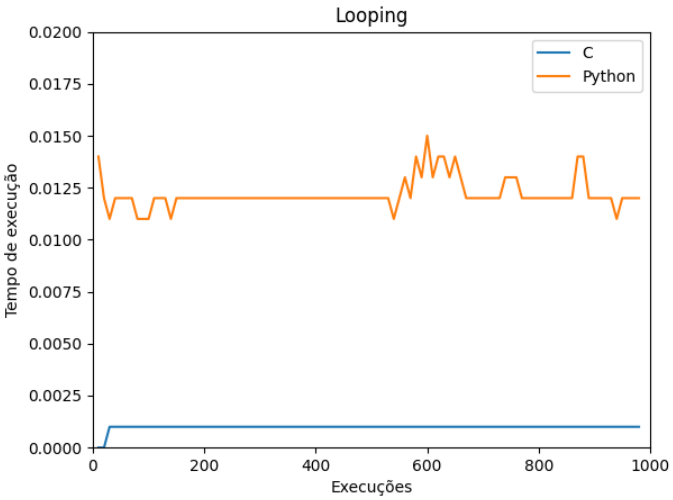

Figure 11: Tempo de execução laço

E podemos ver uma semelhança com o tempo de execução do algortimo de recursão em cauda, como mostra a figura 11. E mais uma vez, podemos comprovar a potência da recursão em cauda.

4 Pontos positivos e negativos

Positivo

- Recursão em cauda sempre é mais rapida que a recursão comum.
- O laço de repetição é uma das estrategias equivalente a recursão em cauda em performance (como foi mostrado teste).

Negativo

- A recursão comum pode estourar a memoria da pilha, devido a necessidade de salvar o endereço para a ultima operação.
- A implementação de recursão em cauda pode ser confusão e complexa.
- A repetição por laço não é um recurso disponibilizado por todos paradigmas.

5 Conclusão

Após a realização deste diversos experimentos, podemos verificar que o paradigma imperativo por ser baseado na arquitetura de Von Neumann, possui um desempenho superior em todos os experimentos. Considerando o cenario de recursão em cauda e loop, o tempo de execução é proximo em todos os casos.
Quando consideramos a recursão comum, e a desvantagem de tal estrategia, o paradigma imperativo consegue uma performance superior aos outro paradigmas pelo fato da sua arquitetura, mas, ainda pode acontecer um extrapolamento da memoria reservada para a pilha.

6 Referencias

Haskell Wiki. Wiki: Introduction. Disponível em: http://www.haskell.org/haskellwiki/Introduction

Python Brasil. Historia do Python. Disponível em: https://wiki.python.org.br/HistoriaDoPython

Alura. Programação funcional. Disponível em: https://www.alura.com.br/artigos/programacao-funcional-o-que-e

Decom-UFOP. Recursividade. Disponível em: http://www.decom.ufop.br/romildo/2012-1/bcc222/slides/06-recursividade.pdf

Matplotlib. Pyplot tutorial. Disponível em: https://matplotlib.org/stable/tutorials/introductory/pyplot.html