Silveira, Paulo António Enes, 1955-

Aprender a melhor programar computadores : com métodos e ferramentas que permitem conceber, executar e testar programas : casos de programação interaciva e recursiva

http://hdl.handle.net/11067/6021
https://doi.org/10.34628/0zy6-we77

APRENDER A MELHOR PROGRAMAR COMPUTADORES:
COM MÉTODOS E FERRAMENTAS QUE PERMITEM CONCEBER, EXECUTAR E TESTAR PROGRAMAS – CASOS DE PROGRAMAÇÃO ITERATIVA E RECURSIVA

LEARN TO PROGRAM COMPUTERS BETTER:
WITH METHODS AND TOOLS THAT ALLOW THE DESIGN, EXECUTION AND TESTING OF PROGRAMS – CASES OF ITERATIVE AND RECURSIVE PROGRAMMING

Paulo Enes Silveira
Universidade Lusíada
Orcid: 0000-0002-9169-8226
paulosilveira@edu.ulusiada.pt

DOI: https://doi.org/10.34628/0zy6-we77
Data de submissão / Submission date: 4 de fevereiro de 2021
Data de aprovação / Acceptance date: 14 de junho de 2021

Resumo: Como conceber e codificar um bom Programa para Programação de Computadores? Esta é uma questão sempre presente na mente de quem aprende e ensina Programação. Perante um Problema a resolver, o presente artigo apresenta métodos para conceber soluções de Programação, apresentando estudos de casos com a aplicação de métodos de Programação Iterativa e Recursiva, tanto na concepção, como na codificação, execução e testes de Programas, num ambiente de aprendizagem interactiva. A Linguagem de Programação Python foi justificadamente escolhida, na sua componente de paradigma imperativo, e aplicou-se a ferramenta Online Python Tutor, para codificação, execução e teste, a qual permite, de forma interactiva, compreender a execução do programa e observar o estado das estruturas de dados envolvidas, passo a passo.

Palavras-chave: Métodos de programação; Programação iterativa; Programação recursiva; Aprendizagem interactiva; Python.

Abstract: How to design and code a good Program for Computer Programming? This is a question that is always present in the mind of those who learn and teach Programming. Facing a Problem to be solved, this article presents methods to design Programming solutions, presenting case studies with the application of Iterative and Recursive Programming methods, both in the design, as in the coding, execution and testing of Programs, in a learning environment interactive. The Python Programming Language was justifiably chosen, in its imperative paradigm component, and the Online Python Tutor tool was applied for coding, execution and testing, which allows, in an interactive way, to understand the execution of the program and observe the state of the data structures involved, step by step.

Keywords: Programming methods; Iterative programming; Recursive programming; Interactive learning; Python.

1. Introdução

Há cerca de setenta anos que se programam computadores. É um curto espaço de tempo no conhecimento humano, mas entre a Programação com sequências de 1 e 0 da linguagem binária, as linguagens de baixo nível e as de elevado nível, desde o paradigma de programação Imperativo ao Paralelo, passando pelo Funcional, pelo Lógico, pelo Orientado por Objectos, entre outros, a Programação teve um desenvolvimento extraordinário, estando hoje presente na generalidade da acção humana assistida pela técnica.

O autor do presente artigo teve o privilégio de aprender e ensinar a Programação de Computadores, durante os últimos quarenta anos e sempre se colocou a mesma questão: Como conceber um bom programa que resolva o desejado?

É uma questão a que os grandes gurus da Programação sempre procuraram responder. E procuraram respostas através de várias abordagens, nomeadamente em livros fundamentais, desde de Donald Knuth considerar a Programação uma Arte (Knuth, D. 1968), Edsger Dijkstra como uma Disciplina (Dijkstra, E. 1976) e David Greis como uma Ciência (Greis, D. 1981). Jacques Arsac foi dos que mais se destacaram, preocupando-se com os mais jovens e os que davam os primeiros passos na Programação, dedicando livros às primeiras lições de Programação (Arsac, J. 1980) e às bases da Programação (Arsac, J. 1983).

Esta questão – Como conceber um bom programa que resolva o desejado? – foi abordada com detalhe pelo autor em (Silveira, P. E. 2016) e (Silveira P. E. & Neves, R. 2019). Convém ter presentes abordagens então feitas: Como resolver um problema em programação em três passos - Interpretar, Formalizar e Resolver; Utilizar a noção de Caixa Preta (contendo a solução), com os resultados pretendidos como saídas e os necessários dados como entradas; as instruções fundamentais do Paradigma Imperativo (atribuição, selecção e ciclo); as possíveis soluções com técnicas típicas de programação - Programação Directa, Iterativa e Recursiva - sendo as duas últimas desenvolvidas no presente artigo.

No ponto 2 volta-se **à questão do parágrafo anterior:** como conceber um bom programa? Nos pontos 3 e 4, abordam-se soluções de programação iterativas e recursivas, salientando-se a importância do exercício da execução do Programa e testes feitos de forma interactiva, o que potencia a aprendizagem, com recurso à Linguagem de Programação Python, justificando-se esta escolha.

Utiliza-se o Python Tutor, uma ferramenta disponível via Internet que permite escrever código em Python e executá-lo passo a passo, visualizando as variáveis utilizadas e a evolução dos seus valores. Por fim, no ponto 5, apresentam-se as conclusões.

2. Resolver Problemas em Programação

Quando se deseja resolver um Problema com Programação, a primeira dificuldade com que nos deparamos é a de observarmos se somos capazes de Interpretar bem o Problema. Interpretar correctamente um Problema exige o conhecimento do seu domínio de aplicação, identificando os resultados que se pretendem atingir e os dados necessários que devem ser considerados para a resolução do mesmo.

Como consequência da interpretação, podemos então Formalizar o Problema, recorrendo à noção de "Caixa Preta" que contém a solução para o Problema – o Algoritmo – colocando à saída da Caixa Preta os resultados e, à sua entrada, os dados de entrada.

Concluídas estas duas fases – Interpretar e Formalizar – o Problema está identificado e podemos agora avançar para o Resolver com Programação.

Resolver um Problema com Programação implica, em primeiro lugar, decidir que paradigma (ou paradigmas) de programação utilizar, pois a solução do Algoritmo de resolução pode ser diversa conforme o paradigma em causa. Esta decisão requer um conhecimento mais avançado da Programação, o qual não iremos considerar no presente artigo. Vamos seguir o caminho indicado por Alan Turing (Turing, A. 1936) com a sua "Universal Computing Machine", a "máquina de Turing", que executa qualquer sequência computável, princípio do paradigma Imperativo.

Utilizaremos o paradigma Imperativo com a noção de Variável e as suas três instruções fundamentais: a Atribuição, a Selecção e o Ciclo, junto com as instruções de entrada e saída de dados, e com os módulos procedimento ou função.

O presente ponto 2. encontra-se desenvolvido pelo autor, com detalhe, em (Silveira, P. E. 2016, pp. 27-34), já referido no ponto 1, com acesso digital em nota de rodapé (leitura aconselhável).

3. Programação Iterativa: Concepção, codificação, execução e testes

3.1 Concepção de Programas Iterativos

A concepção do Algoritmo que resolva o Problema, em particular, no para-

digma Imperativo, deve apoiar-se nas fases de Interpretar e Formalizar este Problema, nas quais devemos poder vislumbrar as ferramentas e técnicas a utilizar (Variáveis; Programação Directa; Iterativa ou Recursiva, entre outras).

Se as sequências de acções do Algoritmo se orientarem para uma sequência de perguntas (selecções) e consequente atribuição de variáveis que constroem os resultados, utilizamos uma técnica de Programação Directa, baseada em selecções e atribuições, como foi exemplificado com a resolução do Problema - Pretende saber-se se uma data é "válida" ou "inválida", em (Silveira, P. E. 2016, pp. 34-39).

Se as acções do Algoritmo necessitarem de recorrer variadas vezes a um mesmo conjunto de instruções para se construírem os resultados, podemos utilizar ciclos da técnica Programação Iterativa ou funções recursivas da técnica Programação Recursiva.

Em (Silveira, P. E. 2016, pp. 39-45), abordámos a resolução do seguinte Problema - Um certo número de facturas devem ser pagas. Dispõe-se de um montante em euros para pagar essas facturas ou parte delas. Quantas facturas podem ser pagas com este montante? A solução então encontrada utilizou um ciclo com algumas selecções que permitiu obter a resposta.

Vamos, de seguida, considerar um Problema um pouco mais exigente na solução a ser encontrada.

PROBLEMA: Deseja-se testar o sabor de um novo iogurte em vários pontos do país. Em cada ponto fazem-se provas com apreciação de 1 voto (mau) a 5 votos (muito bom). Qual o ponto do país que obteve a maior votação?

FASES DE RESOLUÇÃO:

INTERPRETAR: Consideremos **n pontos** no país. Em cada ponto do país fazem-se **n provas**. As classificações de apreciação estão contidas na matriz **miog**, com **n provas** associadas às linhas e **n pontos** associados às colunas. O Algoritmo "Teste de Iogurtes" deve calcular o ponto **maxponto** que tem a maior votação **maxvotos**, dos votos contidos em **miog**.

FORMALIZAR:

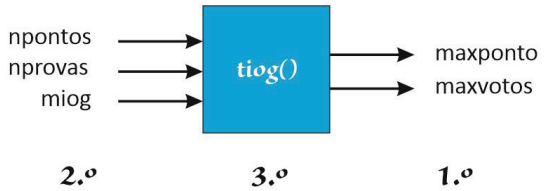

Fig. 1 – Formalizar "Caixa Preta" do tiog() - Teste de Iogurtes

RESOLVER: Vamos considerar dados os valores de entrada.
n pontos = 5; **n provas** = 3;
matriz com 3 linhas de provas e 5 colunas de pontos

miog = [[2,5,4,3,5],
[5,4,2,4,3],
[3,1,3,2,4]].

Observemos que será necessário percorrer todos os elementos da matriz **miog**, a fim de serem consideradas todas as provas realizadas nos diferentes pontos. O tratamento sequencial dos elementos na matriz pode ser feito considerando um atravessamento por linhas (associadas a cada prova feita nos diferentes pontos), ou por colunas (associadas a cada ponto com as provas realizadas). Esta é uma decisão que, tipicamente, suscita dúvidas aos que iniciam o estudo da Programação. Por vezes, esta ordem de atravessamento é indiferente, como seria no caso de o objectivo ser o de se calcular o somatório de todas as classificações obtidas. Mas, no caso particular deste problema **–** o de se saber qual o ponto que reunirá uma maior classificação de provas **–** a ordem de atravessamento já não é indiferente.

Para cada ponto, será necessário calcular o somatório dos votos de cada prova feita nesse ponto, para o poder comparar com o somatório dos outros pontos aferidos, e assim determinar se é ou não um **maxponto** (ponto com maior votação **maxvotos**). Uma vez que as colunas da matriz estão associadas aos pontos, logo, o atravessamento desta deve ser feito por colunas.

Em Programação, o atravessamento sequencial dos elementos de uma matriz é um percurso iterativo através das suas linhas ou colunas indexadas por um índice numérico (um inteiro), sendo o **ciclo** a instrução de Programação que melhor serve este propósito.

Avancemos então na descoberta do necessário algoritmo, seguindo o método de construção de ciclos, em quatro passos, exposto com detalhe em (Silveira, P. E. 2016, pp. 39-45), referido no ponto 1.

No primeiro passo, vamos caracterizar uma "situação geral" do cálculo necessário aos resultados: **maxponto** e **maxvotos.** Consideramos, o número **j-1** de colunas (correspondentes aos pontos) já tratados, conhecendo até então o **maxponto** (ponto com maior votação) e **maxvotos** (número de votos deste ponto) **(1)**.

No segundo passo, verificamos se já se chegou ao fim do tratamento das colunas (se não há mais pontos de provas a considerar). No caso verdadeiro, termina-se o ciclo, chegando-se assim à "Situação Final" do ciclo **(2**). Este ponto de paragem da execução do ciclo é fundamental, para que o ciclo seja finito.

No terceiro passo, avança-se para a "Situação Final" e refaz-se a "Situação Geral", tratando a coluna **j** (correspondente ao ponto **j**), com o cálculo do somatório dos votos das provas feitas **–** o **total –** pelo que será necessário percorrer toda

a coluna **j**, acumulando em **total** todas as votações das **i** provas feitas nesse ponto **j**. Estamos perante a necessidade de um novo ciclo para tratar todas as provas de ordem **i**, o qual consideramos uma nova Caixa Preta a resolver, após termos concluído a escrita do ciclo exterior **j**. Desta Caixa Preta resulta o **total** dos votos do ponto **j** que deve ser comparado com o anterior **maxvotos,** para sabermos se foi ou não foi encontrado um novo **maxponto** (SE **total>maxpontos** ENTÃO **maxvotos** ← **total**; **maxponto** ← **j** ES). Refaz-se a "Situação Geral" com o incremento de 1 do parâmetro iterativo j (j ← j + 1), preparando assim a próxima iteração **(3)**.

Por fim, no quarto passo, inicializam-se as necessárias variáveis (antes da Situação Geral). Considera-se o parâmetro iterativo **j** iniciado com o valor zero (j ← 0), tendo em conta que na Linguagem de Programação que se utilizar, o primeiro índice de elemento de linha ou coluna da tabela-matriz é zero; o futuro resultado de **maxvotos** também deve ser inicializado com zero (maxvotos ← 0) **(4)**.

Resumimos de seguida os quatro pontos da construção do ciclo j, com um PSEUDOCÓDIGO, em que usamos o caracter # para assinalar comentários.

```
# PASSO QUATRO
    j ← 0
    maxvotos ← 0
                                                    (4)
# PASSO UM
    FAZER # o ciclo j
    # SITUAÇÃO GERAL:  Consideramos o número j-1 de colunas (correspondentes aos
    # pontos) já tratados, conhecendo até então o maxponto (ponto com maior votação) e
    # maxvotos (número de votos deste ponto)
                                                    (1)
# PASSO DOIS
    # Verificar se já chegou ao fim
        SE j > npontos-1 ENTÃO SAIR ES # sair terminando o ciclo j
        ES
                                                    (2)
# PASSO TRÊS
    # Avançar para a Situação Final e refazer a Situação Geral
        # CAIXA PRETA para cálculo do total de votos no ponto j
        SE total > maxvotos ENTÃO
            maxvotos ← total
            maxponto ← j
        ES
        j ← j + 1
```

```
        REFAZER # FAZER o próximo passo do ciclo j
```
(3)

Falta dar solução à CAIXA PRETA para cálculo do **total** de votos no ponto **j**, desenvolvendo o ciclo **i** que percorre todas as **nprovas** com os seus votos. O método de construção do ciclo é o mesmo que aplicámos ao ciclo **j (5)(6)(7)(8)**.

```
# PASSO QUATRO
    i ← 0
    total ← 0
```
(8)

```
# PASSO UM
    FAZER # o ciclo i
    # SITUAÇÃO GERAL:  Consideramos o número i-1 de linhas (corres-
    pondentes às
    # provas) já tratadas, conhecendo até então o total de votos do ponto j
```
(5)

```
# PASSO DOIS
    # Verificar se já chegou ao fim
        SE i > nprovas-1 ENTÃO SAIR ES # sair terminando o ciclo i
        ES
```
(6)

```
# PASSO TRÊS
    # Avançar para a Situação Final e refazer a Situação Geral
        total ← total + miogi,j
        i ← i + 1
```
(7)

```
    REFAZER # FAZER o próximo passo do ciclo i
```

Construídos os dois ciclos, obtemos o algoritmo que resolve este problema, obtendo os resultados **maxponto** e **maxvotos**:

```
j ← 0
maxvotos ← 0
FAZER # o ciclo j
    SE j > npontos-1 ENTÃO SAIR ES # sair terminando o ciclo j
    ES
    i ← 0
    total ← 0
    FAZER # o ciclo i
        SE i > nprovas-1 ENTÃO SAIR ES # sair terminando o ciclo i
```

```
            ES
            total ← total + miogi,j
            i ← i + 1
        REFAZER # FAZER o próximo passo do ciclo i
        SE total > maxvotos ENTÃO
            maxvotos ← total
            maxponto ← j
        ES
        j ← j + 1
    REFAZER # FAZER o próximo passo do ciclo j
```

(9)

Concluída a fase de Resolver o Problema, temos o Algoritmo de resolução. Segue-se a escolha de uma Linguagem de Programação para a sua codificação, execução e teste.

3.2 Codificação, execução e testes de Programas Iterativos

A Linguagem de Programação Python foi a escolhida com base na tendência dos últimos seis anos, seja a nível do ensino na introdução à Programação, seja na sua utilização a nível mundial.

Em 2014, Philip Guo publicou no Blog do ACM-Association Computing Machinery o estudo "Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities" (Philip, G. 2014) , onde verificou que em 8 dos 10 Departamentos de *Computer Science* (80%) e em 27 de 39 (69%) das Universidades de topo dos Estados Unidos da América, Python era a eleita para o ensino introdutório da Programação. Em 2015, Esther Shein salienta na secção de notícias da Communications of ACM, citando este mesmo estudo, que a linguagem Python está a ser usada nas disciplinas de Introdução à Programação, nas Universidades que ensinam Ciência dos Computadores (Shein, E. 2015).

Desde 2017 que a revista Spectrum IEEE, vem anunciando que a linguagem Pyhton alcançou o primeiro lugar, destronando outras linguagens como Java, C e C++. Stephen Cass anunciou na Spectrum IEEE, em Julho de 2020, que Python continua a dominar na "Top Programming Languages 2020" desta revista, aparecendo em primeiro lugar, de novo à frente de Java, C e C++, numa lista das 55 linguagens mais populares, baseada em 11 métricas verificadas através de ferramentas digitais (Cass, S. 2020).

Vamos, então, codificar o Algoritmo do Teste de Iogurtes, concebido em 3.1, com a linguagem Python.

Tradicionalmente, seria exigido ao estudante de programação que tivesse ao seu dispor um ambiente de programação de Python, tal como o que se pode obter em www.python.org, ou ambientes de desenvolvimento (*IDE-Integrated Development environment*), como *o Spyder-The Scientific Python Development Environment* da Anaconda ou o *PyCharm* da JetBrains; que o configurasse adequadamente no seu sistema operativo; que tivesse uma aprendizagem introdutória a estes mesmos ambientes. Enfim, procedimentos que são desejáveis e necessários para uma fase mais avançada da aprendizagem da Programação (permitindo total acesso às potencialidades da linguagem e dos seus ambientes de desenvolvimento), mas que consomem tempo e esforço.

Propomos um caminho bem mais facilitado e que permite a codificação, a execução e testes do Programa, com o Online Python Tutor (Guo, P. 2013) (Guo, P. & Markel, J. & Zhang, X. 2020), concebido por Philip Guo, em ambiente *online* gratuito, educacional, que permite as seguintes funcionalidades:

a) Utilização através de um *browser* da Web**,** sem *plugins*, ou outras configurações (mesmo em dispositivos móveis);
b) Codificação interactiva em Python, em janela de edição;
c) Execução do código com possibilidade de passo a passo para a frente e para trás;
d) Janela de saída de resultados;
e) Visualização do estado das estruturas de dados do programa, nomeadamente na execução passo a passo;
f) Permite o teste do programa com alteração interactiva do código;
g) Permite a geração de um link permanente ao programa escrito, partilhável;
h) Dispõe cerca de 5 dezenas de exemplos de código em Python;
i) Este Online Tutor inclui outras linguagens: Java, C, C++, JavaScript e Ruby;
j) Dispõe acesso a *Help* (Ajuda) através da larga comunidade PYTHON DISCORD.

Para além do acesso imediato à codificação, execução e teste, o Python Tutor tem uma mais valia fundamental, a de se seguir a evolução do estado das estruturas de dados, observando-se os valores intermédios das mesmas até aos resultados finais, de forma automática, facilitando a compreensão das acções do programa. Esta tarefa é muitas vezes feita com recurso a outros suportes (papel ou quadro), simulando a execução do programa, pois esta é uma das dificuldades a serem ultrapassadas pelos estudantes de Programação – a verificação do que faz um Programa, passo a passo.

Na codificação do Algoritmo "Teste de Iogurtes" em Python, apresentado em **(9)**, transformamos os dois ciclos FAZER…REFAZER**(10)** com inicialização prévia dos parâmetros iterativos **j** e **i**, a sua incrementação e a condição de saída dos mesmos ciclos, em dois ciclos equivalentes **for** do Python **(11)**:

```
j ← 0
FAZER # o ciclo j
    SE j > npontos-1 ENTÃO SAIR ES # sair terminando o ciclo j
    ES
    i ← 0
    FAZER # o ciclo i
        SE i > nprovas-1 ENTÃO SAIR ES # sair terminando o ciclo i
        ES
        # …
    REFAZER # FAZER o próximo passo do ciclo i
    j ← j + 1
REFAZER # FAZER o próximo passo do ciclo j
```

(10)

```
for j in range(0, npontos):

for i in range(0, nprovas):
```

(11)

O restante Algoritmo é codificado directamente em Python, na janela-editor do Online Python Tutor. Na Fig. 2, efectuou-se uma execução passo a passo, estando representado o passo 17 dos 61 totais que o programa efectua, e o estado das estruturas de dados (variáveis e lista de listas representando a matriz **miog**) com os respectivos valores.

Observa-se que no passo 17, o ciclo for **j** foi executado uma vez (**j=0**); o ciclo interior **i** foi executado de i = 0 até i = nprovas-1 (**i=2**); o **total** acumulou os valores de votos 2, 5 e 3 (**total=10**); Na linha 11 verifica-se que **total > maxvotos**, tendo-se encontrado que o ponto j=0 é agora o novo **maxponto=0** com **maxvotos=10** (linhas 12 e 13).

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  npontos=5 # número de Pontos de Provas
2  nprovas=3 # número de Provas em cada Ponto
3  miog=[[2,5,4,3,5],[5,4,2,4,3],[3,1,3,2,4]] # inicia a matriz
4  maxvotos=0
5  for j in range(0,npontos):
6      # ciclo j para as colunas da matriz miog
7      total=0
8      for i in range(0,nprovas):
9          # ciclo i para as linhas da matriz miog
10         total = total + miog[i][j]
11     if (total > maxvotos):
12         maxvotos=total
13         maxponto=j
14 print('O Ponto que teve mais votos é:',maxponto)
15 print('com o número de votos =',maxvotos)
```

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 17 of 61

Print output (drag lower right corner to resize)

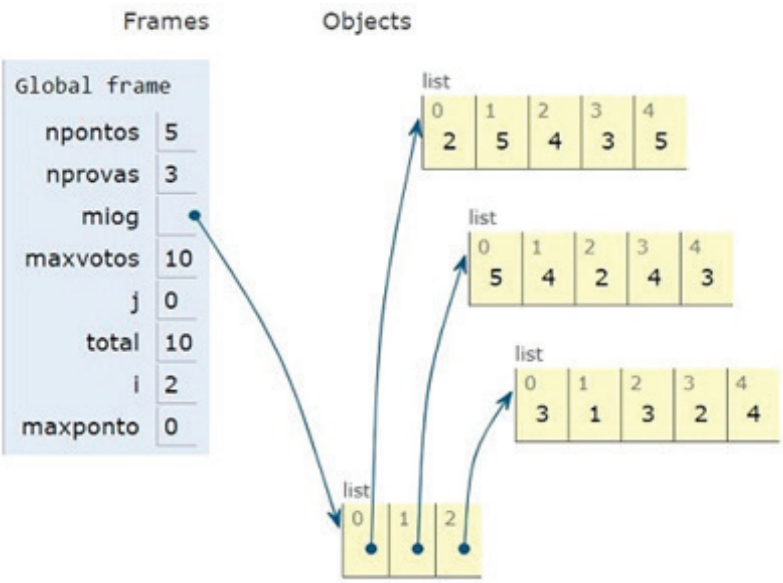

Fig. 2 – Execução do passo 17 do programa "Teste de Iogurtes", no Python Tutor

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  npontos=5 # número de Pontos de Provas
2  nprovas=3 # número de Provas em cada Ponto
3  miog=[[2,5,4,3,5],[5,4,2,4,3],[3,1,3,2,4]] # inicia a matriz
4  maxvotos=0
5  for j in range(0,npontos):
6      # ciclo j para as colunas da matriz miog
7      total=0
8      for i in range(0,nprovas):
9          # ciclo i para as linhas da matriz miog
10         total = total + miog[i][j]
11     if (total > maxvotos):
12         maxvotos=total
13         maxponto=j
14 print('O Ponto que teve mais votos é:',maxponto)
15 print('com o número de votos =',maxvotos)
```

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Done running (61 steps)

Print output (drag lower right corner to resize)

```
O Ponto que teve mais votos é: 4
com o número de votos = 12
```

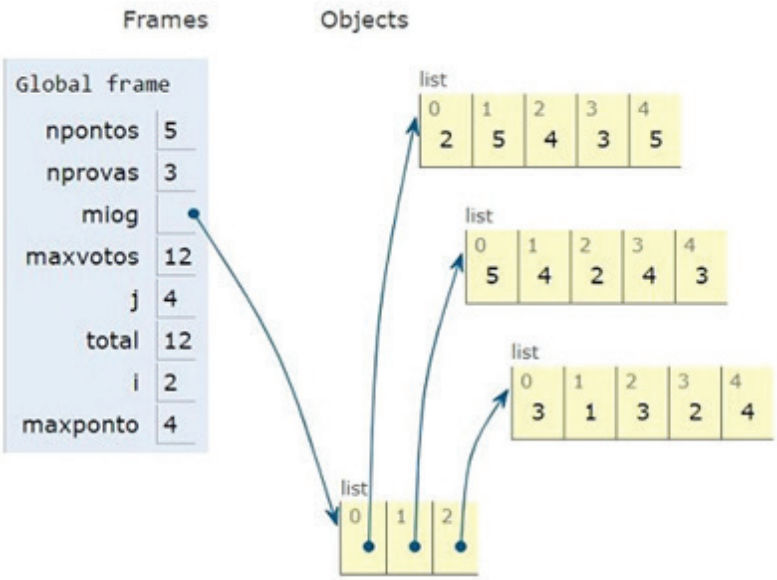

Fig. 3 – Execução final do programa "Teste de Iogurtes", no Python Tutor

A Fig. 3, mostra a situação final do programa, após 61 passos, com os resultados afixados de **maxponto=4** (o ponto mais votado correspondente à quinta coluna da matriz miog) e de **maxvotos=12** (os votos do ponto mais votado). Também se podem constatar os valores finais das várias variáveis usadas.

Para que possam fazer a experiência de utilização do Python Tutor, basta aceder a www.pythontutor.com e usar o código em causa (12).

```
npontos=5 # número de Pontos de Provas
nprovas=3 # número de Provas em cada Ponto
miog=[[2,5,4,3,5],[5,4,2,4,3],[3,1,3,2,4]] # inicia a matriz
maxvotos=0
for j in range(0,npontos): # ciclo j para as colunas da matriz miog
   total=0
   for i in range(0,nprovas): # ciclo i para as linhas da matriz miog
      total = total + miog[i][j]
   if (total > maxvotos):
      maxvotos=total
      maxponto=j
print('O Ponto que teve mais votos é : ',maxponto)
print('com o número de votos = ',maxvotos)
```

(12)

No Python Tutor a alteração do código pode ser feita interactivamente. Uma vez feita a alteração, basta realizar uma nova execução. Sugerimos que o Algoritmo "Teste de Iogurtes" deva também calcular o ponto **minponto** que tem a menor votação **minvotos**, dos votos contidos em **miog**. Basta acrescentar mais dois dados de saída à Caixa Preta da Fig. 1: o ponto **minponto** com a sua menor votação **minvotos**, alterar o algoritmo e codificá-lo. Dado que o processo de construção dos resultados **maxvotos** e **maxponto** foi já estudado, esta alteração é simples de realizar directamente no código **(12)** já escrito no Python Tutor, acrescentando o cálculo de **minvotos** e **minponto**.

4. Programação Recursiva: Concepção, codificação, execução e testes

4.1 Concepção de Programas Recursivos

A Programação Recursiva tem uma beleza especial. Recorre a um mesmo módulo de instruções para obter o resultado desejado, através de um processo de chamada a esse próprio módulo, seja uma função que se invoca a si própria. Parece o processo mágico da reflexão da nossa imagem em dois espelhos parale-

los, em que um espelho reflete a nossa imagem no outro, dando a ilusão de uma reflexão infinita.

A Programação Iterativa abordada no ponto 3. também faz repetir um conjunto de instruções até uma situação de paragem identificada, indicando que o ciclo da repetição já concluiu o seu trabalho. Este conjunto de instruções está totalmente à vista do programador.

A recursividade em Programação tem de ser finita, através da sua situação de paragem que termina a chamada recursiva. Mas o conjunto de instruções que chama recursivamente, é habitualmente pequeno e nem sempre está claramente à vista do Programador. Este facto dificulta a aprendizagem de um estudante de Programação.

Numa comparação metafórica, a Programação Iterativa está para a Programação Recursiva, como **–** uma escada com o seu patamar inicial, os seus degraus bem à vista e o seu patamar superior (ou inferior) **–** está para **–** uma cebola, com a sua camada exterior e as sucessivas camadas interiores até à última no seu núcleo central. A diferença entre as duas evidencia-se do ponto de vista do observador (programador). Na Iterativa, está tudo visível, do patamar inicial ao final da escada, cada degrau é uma iteração. Na Recursiva, a cebola inteira vista de fora nada revela do seu interior, mas se a cortarmos ao meio, logo vislumbramos as várias camadas concêntricas, do exterior para o interior, correspondendo cada salto de camada a uma chamada recursiva. Numa outra aproximação, a Iteraviva está para a Recursiva como uma prosa está uma poesia. A prosa é habitualmente descritiva e longa, a poesia é algo enigmática e curta, mas quando bem feita é sempre bela.

Vamos procurar desmontar esta dificuldade e apontar caminhos da construção da Recursividade em Programação, na senda do que já foi abordado por Jacques Arsac nos seus livros *Préceptes pour Programmer* (Arsac, J. 1991) e *Programação Criativa, Preceitos e Instrumentos* (Arsac, J., 1993).

Consideremos um Problema simples, para nos concentrarmos no método de o resolver recursivamente.

PROBLEMA: Calcular a potência de um número inteiro elevado a outro número inteiro positivo, utilizando uma função recursiva.

FASES DE RESOLUÇÃO:

INTERPRETAR: Seja a função pot(a,p). Devemos conhecer os dois parâmetros da função: Os valores da base **a** (número inteiro) e da potência **p** (número inteiro positivo). A função pot(a,p) deve calcular ap . O resultado será um número inteiro.

FORMALIZAR:

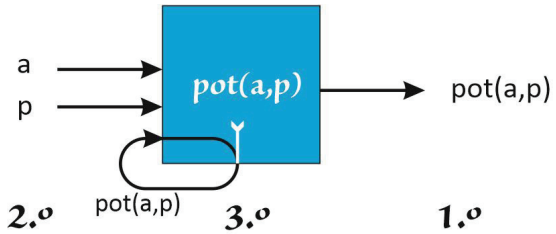

Fig. 4 – Formalizar "Caixa Preta" de pot(a,p) recursiva

RESOLVER: Como criar a função recursiva pot(a,p)?

Se não temos ideia ou orientação alguma, vamos manipular um exemplo **(13)**.

$$\text{pot}(2,3) = 2 \cdot 2 \cdot 2 \qquad \textbf{(13)}$$

Agora estamos no momento de sermos criativos. Desta expressão **(13)** como fazer surgir uma chamada recursiva à própria função? Que tal um truque de magia?

Experimentemos a enquadrar uma parte da definição extensiva 2 · 2 · 2, deixando de fora o primeiro factor 2 **(14)**, de forma a evidenciar uma definição recursiva.

$$\text{pot}(2,3) = 2 \cdot \underline{\mathbf{2 \cdot 2}} \qquad \textbf{(14)}$$

Como **2 · 2** é a definição extensiva de pot(2,2), feita esta substituição, está descoberta a definição recursiva **(15)**!

$$\text{pot}(2,3) = 2 \text{ x } \underline{\mathbf{pot(2,2)}} \qquad \textbf{(14)}$$

Generalizando, façamos o mesmo para pot(a,p), aprimorando o método.

No primeiro passo, apresentar uma definição extensiva da função **(15)**.

$$\text{pot}(a,p) = \underbrace{a \cdot a \cdot \ldots \cdot a \cdot a}_{p \text{ vezes}} \qquad \textbf{(15)}$$

No segundo passo, enquadrar uma parte da definição, de forma a evidenciar uma definição recursiva **(16)**.

$$\text{pot}(a,p) = \underbrace{a \cdot \mathbf{a} \cdot \ldots \cdot \mathbf{a} \cdot \mathbf{a}}_{\text{p-1 vezes}} \tag{16}$$

No terceiro passo, substituir a parte enquadrada pela sua equivalente função pot **(17)**.

$$\text{pot}(a,p) = a \cdot \text{pot}(a,p-1) \tag{17}$$

No quarto passo, identificar a situação de paragem da recursividade **(18)**. Esta deve resultar de uma situação conhecida na evolução das chamadas recursivas.

$$\text{SE } p = 1 \text{ ENTÃO RESULTA } a \text{ ES} \tag{18}$$

Deduzida a definição recursiva e a sua situação de paragem, já podemos apresentar o Algoritmo da função recursiva **(19)**.

$$\begin{aligned} \text{pot}(a,p) \approx\ & \text{SE } p = 1 \text{ ENTÃO RESULTA } a \\ & \qquad \text{SENÃO RESULTA } a \cdot \text{pot}(a,p-1) \\ & \text{ES} \end{aligned} \tag{19}$$

A esta solução chamo um belo poema de programação. Uma pergunta para sabermos se acaba e no caso contrário, uma chamada recursiva. É simples, elegante e encerra em si alguma magia. A sua compreensão, por parte de estudantes iniciados na Programação, não é simples. A primeira leitura da função leva-nos a constatar que se **p>1**, há lugar a chamadas recursivas sucessivas, que acabam quando **p=1**, resultando **a**. Logo, parece poder concluir-se que o resultado da função **pot** é sempre **a**. É esta a primeira dificuldade a contornar-se.

Sabemos que 2 elevado a 3 resulta 8. Mas da leitura feita, pot(2,3) pareceu resultar 2 (a=2, na situação de paragem da recursividade). Então, há algo que se passa para lá da leitura feita – a tal magia.

Na realidade, uma função recursiva em programação pode ter algo escondido, que se passa sem o programador iniciado se aperceber. Esta função pot(a,p)

tem uma recursividade "não-terminal", uma vez que o cálculo do seu valor não termina com o resultado da situação de paragem. A própria linguagem de Programação, gere a execução do programa, arquivando cada expressão de chamada recursiva numa pilha (*stack*, uma lista LIFO – *Last In First Out* – último a entrar é o primeiro a sair) para posterior uso. Quando a situação de paragem é atingida, o valor resultante é utilizado para ser substituído na última expressão de chamada recursiva carregada na referida pilha, sendo feitas substituições sucessivas até que não existam mais expressões nessa pilha. Observemos um exemplo de execução de pot(2,3) **(20)** e o posterior cálculo, após a situação de paragem **(21)**.

Execução:

Função	RESULTA	Pilha
pot(2,3)	2 · pot(2,2)	
pot(2,2)	2 · pot(2,1)	2 · pot(2,1)
pot(2,1)	2	2 · pot(2,2)

(20)

Cálculo após a situação de paragem:

Função	Pilha	Cálculo
1º pot(2,2) =	2 · pot(2,1)	2 · 2 = 4
2º pot(2,3) =	2 · pot(2,2)	2 · 4 = 8 Resultado final

(21)

4.2 Codificação, execução e testes de Programas Recursivos

Utilizemos o Online Python Tutor, que apresentámos em 3.2 para codificar a função recursiva pot(a,p), executar e testar.

Na Fig. 4, apresentamos a execução do passo 15, dos 17 do programa, a=2 e p=3. Na linha 7 observa-se que a situação de paragem foi atingida e o valor de a=2 é o resultado de pot(2,1). Na parte onde se observam as estruturas de dados, as duas chamadas recursivas pot(2,2) e pot(2,1) com "Return Value"=2, aparecem na coluna "Frames".

Na Fig. 5, observa-se o passo 17 que se mantém a execução na linha 7, pois as substituições das expressões das chamadas recursivas estão a ser feitas como é dado ver na coluna "Frames": pot(2,3) com "Return Value"=8.

A Fig. 6 mostra que como já não há mais expressões recursivas na pilha, então 8 é o valor final de pot(2,3) e o resultado é afixado.

Código para executar no Python Tutor **(22)**.

```
def pot(a,p):
    """pot(a,p) é uma função recursiva para calcular
    a potencia de a elevado a p;
    a-base; p-exponente inteiro positivo;
    """
    if p == 1:
        return a
    else:
        return (a*pot(a,p-1))
base = int(input('Valor da base a:'))
potencia = int(input('Valor da potencia p:'))
print(base,'elevado a',potencia,'recursivo =',\
                    pot(base,potencia))
```

(22)

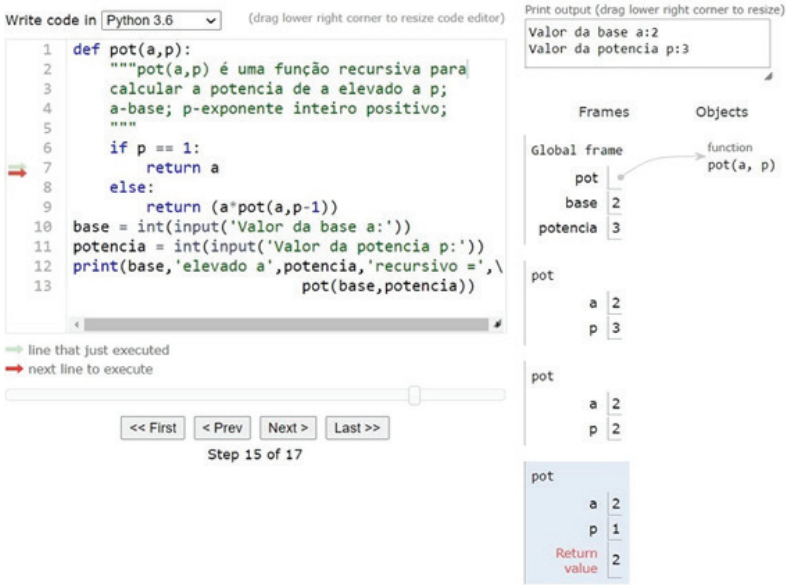

Fig. 5 – Execução do passo 15 do programa recursivo pot(a,p), no Python Tutor

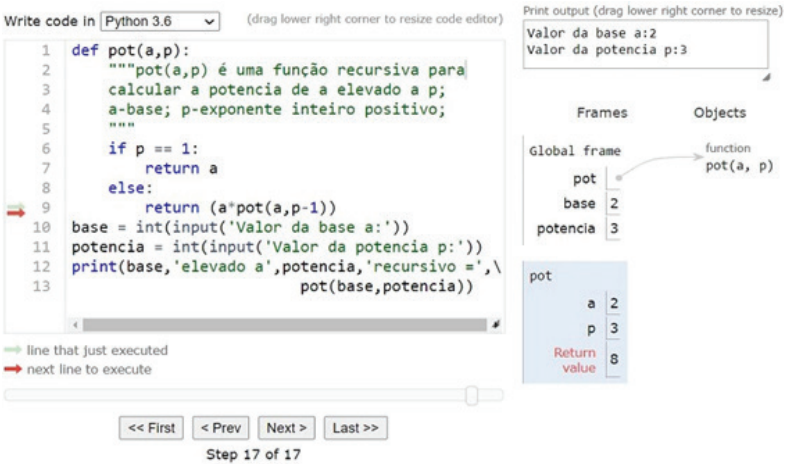

Fig. 6 – Execução do passo 17 do programa recursivo pot(a,p), no Python Tutor

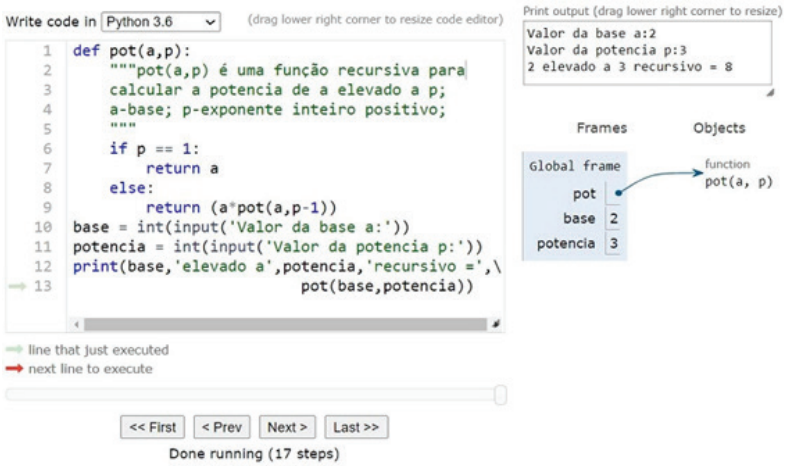

Fig. 7 – Execução final do programa recursivo pot(a,p), no Python Tutor

4.3 O caso da Recursividade Terminal

A maioria das soluções recursivas é "não-terminal", como o caso da exposta no ponto 3. Não terminal, pois a condição de paragem da solução recursiva retorna um valor que não é o final. É automaticamente feita uma substituição deste valor nas expressões guardadas na pilha (*stack*) interna auxiliar, até esta estar vazia, obtendo-se assim o valor final. Neste tipo de recursividade, as expressões das chamadas recursivas contêm operadores e operandos cujo resultado não devolve um valor imediato. Seja o exemplo da função rescursiva **pot(a,p)**, que retorna a expresão: **a · pot(a,p-1)**. Como desta expressão não resulta um valor, mesmo com os valores de **a=2** e **p=3**, o resultado é **2 · pot(2,3)**, logo esta mesma é carregada

na pilha interna desta recursividade, com as consequentes acções já abordadas em 3..

Mas quando a expressão de chamada recursiva devolve um valor imediatamente calculável, o resultado retornado pela situação de paragem, é o valor final. Estamos no caso de uma "recursividade terminal".

Estudemos o tradicional algoritmo de Euclides para o cálculo do máximo divisor comum entre dois números Naturais **m** e **n** com **m>n**, **mdc(m,n) (23)**.

Dividir **m** por **n**, obter o resto **r1**. Se **r1=0**, o **mdc** é o divisor **n**
Se **r1≠0** dividir **n** por **r1**, obter o resto **r2**. Se **r2=0**, o **mdc** é o divisor **r1**

Se **r2≠0** dividir **r2** por **r1**, obter o resto **r3**. Se **r3=0**, o **mdc** é o divisor **r3**
…
Se **rn≠0** dividir **rn-1** por **rn**, obter o resto **rn+1**. Se **rn+1=0**, o **mdc** é o divisor **rn**
(23)

A solução recursiva deste algoritmo, está já contida no mesmo **(24)** (considerando o operador **%** como sendo o resto da divisão inteira entre os operandos **m** e **n)**.

$$mdc(m,n) = mdc(n, m\%n) \quad \textbf{(24)}$$

A situação de paragem é, neste algoritmo, uma situação que conduz ao resultado final **(25)**.

SE m%n = 0 ENTÃO RESULTA <divisor> ES
(25)

A função recursiva **mdcER(m,n)** segundo o algoritmo de Euclides, surge assim com o algoritmo seguinte **(26)**.

mdcER(m,n) ≈ SE n=0 ENTÃO RESULTA m
SENÃO RESULTA mdcER(n,m%n)
ES
(26)

A chamada recursiva feita através da invocação da função mdcER(n,m%n) pode calcular-se e leva-nos para o resultado final, caso o resto calculado seja igual a zero. Estamos perante uma recursividade terminal.

A codificação, execução e teste desta recursividade terminal está na Fig.8, que mostra o passo 15, na linha 3, com o "Return value" terminal, igual a 2, após duas chamadas recursivas, que se podem constatar na coluna "Frames" do Python Tutor.

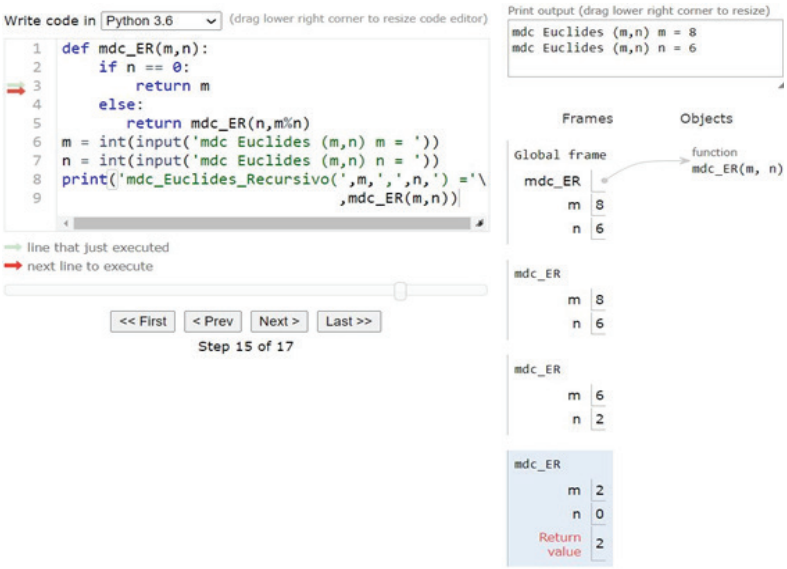

Fig. 8 – Execução do passo 15 do programa que executa mdcER(m,n), no Python Tutor

A Fig.9 mostra a execução final do programa, com a afixação do resultado de mdcER(8,6) = 2. A recursividade terminal obtém o valor final directamente da situação de paragem, sem serem necessários os cálculos com recurso à pilha interna da recursividade não-terminal.

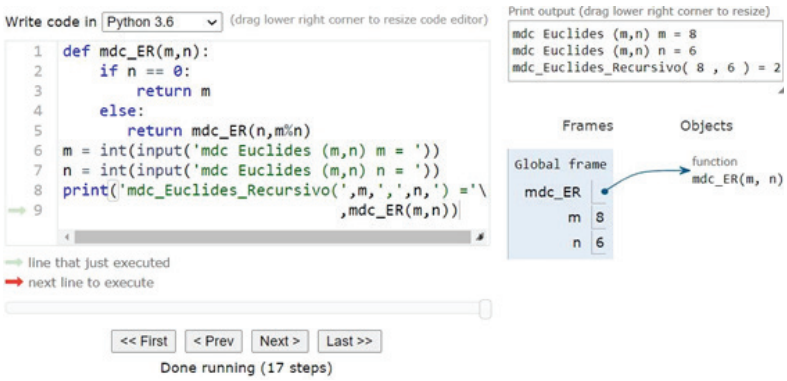

Fig. 9 – Execução final do programa que executa mdcER(m,n), no Python Tutor

5. Conclusões

No presente, aprender a programar parece ser uma tarefa facilitada com o acesso à Internet e às inúmeras lições e tutoriais disponíveis na Web. Mas estes dedicam-se, na sua generalidade, a ensinar linguagens de programação, ou a mostrar como resolver problemas apontando as suas soluções com propostas de código para a resolução. Este artigo tem em conta as dificuldades que os estudantes de programação evidenciam para encontrar uma solução de resolução de um Problema com Programação, apresentando métodos que facilitam a concepção de uma solução. Uma vez na presença de um algoritmo de resolução, há que o codificar, executar e testar, com uma linguagem de programação, compreendendo o que esta executa.

Para a concepção da resolução do Problema, apresentámos métodos que consideram como Interpretar, Formalizar (com a "Caixa Preta", suas saídas e entradas) e Resolver até à obtenção do Algoritmo de resolução. Estudámos alguns casos que permitiram aplicar métodos propostos, seja na construção de algoritmos iterativos, em que a solução utiliza a instrução de programação ciclo, sejam recursivos com a construção de funções recursivas.

Na construção de soluções iterativas ou recursivas, algumas vezes estas soluções surgem-nos com clara evidência, seja por experiência de sucesso anterior idêntica, seja por inspiração especial. Tanto melhor! Mas quando não vislumbramos o caminho a seguir, os métodos apresentados são uma mais valia para o programador, especialmente o estudante iniciado de programação.

Seguiu-se a codificação na Linguagem de Programação. Escolhemos o Python com a sua componente de paradigma imperativo, justificando a escolha. Mostrámos a importância de se compreender a execução do programa através de testes feitos com o Online Ptyhon Tutor, que permite numa aprendizagem interactiva imediata, executando passo a passo (para a frente e para trás) e acompanhando o estado das estruturas de dados manipuladas pelo Programa, o que melhora significativamente a compreensão do mesmo.

Com estas componentes de concepção e de codificação, execução e teste interactivos, temos melhores ferramentas para aprender e ensinar a melhor Programar Computadores.

Agradecimento

O autor agradece reconhecidamente ao Professor Rui Agonia Pereira as inúmeras conversas científicas que, ao longo de muitos anos, ambos vêm dedicando ao tema da Programação, incluindo o presente artigo na sua fase de *working paper*.

Bibliografia

Arsac, J. (1980). *Premières leçons de programmation.* Paris: Cedic.

Arsac, J. (1983). *Les bases de la programmation*. Paris: Dunod.

Arsac, J. (1991). *Préceptes pour Programmer*. Paris: Dunod.

Arsac, J. (1993). *Programação Criativa, Preceitos e Instrumentos*. T. L. Castro (Ed.), Mem Martins, Portugal: CETOP. Prólogo de P. E. Silveira.

Cass, S. (2020) Top Programming Languages 2020, *Spectrum IEEE.*
https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020

Dijkstra, E. (1976). *A Discipline of Programming*. Prentice Hall.

Greis, D. (1981). *The Science of Programming.* New York: Springer Verlag.

Guo, P. (2013) Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. *ACM Technical Symposium on Computer Science Education (SIGCSE),* 2013. Consultado a 12/10/2020: https://dl.acm.org/doi/10.1145/2445196.2445368

Guo, P. (2014) Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities, July 7 2014, bolg@cacm, *Communications of the ACM*, 7 p.
https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext

Guo, P. & Markel, J. & Zhang, X. (2020). Learnersourcing at Scale to Overcome Expert Blind Spots for Introductory Programming: A Three-Year Deployment Study on the Python Tutor Website. *L@S '20: Proceedings of the Seventh ACM Conference on Learning @ Scale* August 2020 pp. 301–304 https://doi.org/10.1145/3386527.3406733

Knuth, D. (1968). *The art of computer programming: Fundamental Algorithms* (3rd. Ed.) vol 1, Addison Wesley.

Shein, E..(2015) Python for Beginners, *Communications of the **ACM***. Mar2015, Vol. 58 Issue 3, p19-21. https://dl.acm.org/doi/10.1145/2716560

Silveira, P. E. (2016). Programação Directa e Iterativa, Métodos para criar bons Programas. Tributo a Jacques Arsac*, Economia & Empresa,* **21**, Lusíada, Lisboa, 21-51 (2016).
http://revistas.lis.ulusiada.pt/index.php/lee/article/view/2362

Silveira, P. E. & Neves, R. (2019). How to achieve better performance in teaching computer programming: Cases of iterative and recursive programming. AIP Conference Proceedings. 2116. 410003. https://doi.org/10.1063/1.5114427

Turing, A. (1936). On Computable Numbers, with an application to the Entscheidungsproblem. In *Proc. Lond. Math. Soc.* (2) 42 230-265. Obtido a 12/10/2020, de https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf; Correction ibid. 43, 544-546 (1937).