**Texto base**

**11**

**Busca e Ordenação**

Gilberto Alves Pereira

***Resumo***

*Neste capítulo vamos abordar algoritmos um pouco mais complexos. Primeiramente vamos discutir dois tipos de busca: a busca linear e uma busca otimizada que é a busca binária. Em seguida vamos discutir um algoritmo de ordenação pouco eficiente mas de fácil implementação: o Bubble Sort. Ao final, mostramos a conversão do Bubble Sort para Python.*

# 1.1. Introdução

A implementação de algoritmos um pouco mais sofisticados ajuda muito na prática e utilização das ferramentas de programação. A seguir vamos abordar três algoritmos: A busca linear, a busca binária e o algoritmo de ordenação Bubble Sort.

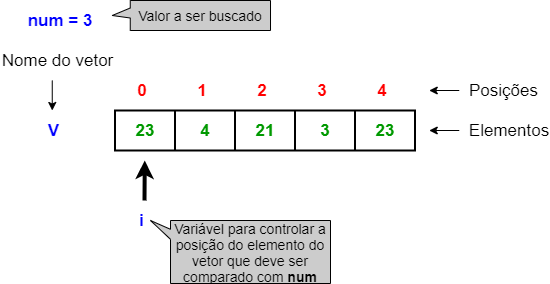
# 1.2. Busca Linear

É muito comum no desenvolvimento de software nos defrontarmos com a necessidade de buscar, pesquisar. Basta entrar em qualquer site que sempre iremos encontrar o desenho de uma lupa. Sem dúvida a pesquisa nos poupa de um gasto enorme de tempo.

A busca linear é um algoritmo básico de busca. Ela é feita a partir de um local onde as informações (normalmente uma grande quantidade) é armazenada. Vamos usar como local de armazenamento vetores. O algoritmo consiste em verificarmos todos os elementos até encontrarmos o que procuramos, ou até verificarmos todos os elementos para constatarmos que ele não foi encontrado.

Vejamos o exemplo 1 abaixo:

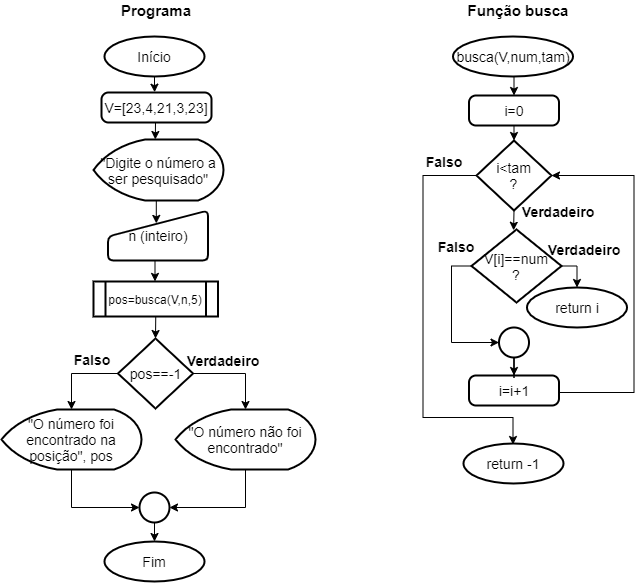
**Exemplo 1** - Criar uma função que receba como parâmetro um vetor com números e um número a ser procurado. A função deve retornar a posição do vetor onde o número foi encontrado ou -1 caso não encontre o número.



**Figura 11.1. Esquema de funcionamento da busca linear.**

**Fonte Autor.**

Basicamente o algoritmo usa uma variável auxiliar **i** para ajudar a percorrer todas as posições do vetor (Figura 11.1) Ou seja haverá uma comparação para cada posição do vetor. Usaremos um loop para isso e uma estrutura de seleção para fazer as comparações. No exemplo da figura 11.1 estamos procurando o número 3 no vetor **V.** A busca inicia-se pela posição 0 (elemento 23) e vai percorrendo o vetor (através do conteúdo da variável **i** ) até encontrar o elemento 3 na posição 3 do vetor. Uma vez encontrado o valor que buscamos não precisamos continuar procurando. Podemos encerrar a busca e retornar o valor 3 (a posição onde o elemento 3 foi encontrado).



**Figura 11.2. Fluxograma da busca linear com programa e função.**

**Fonte Autor.**

Na Figura 11.2 podemos observar o fluxograma da função e do programa que chama a função. No programa devemos criar o vetor **V** com os elementos da base de valores em que será feita a busca e obter do usuário o valor **n** que iremos buscar no vetor. Observe que na chamada da função **pos=busca(V,n,5)** estamos passando três parâmetros: **V** o vetor que possui os dados, **n** o número a ser procurado no vetor e **5** o tamanho do vetor). Além disso, estamos usando a variável **pos** que vai receber o retorno da nossa função **busca**. Caso o retorno seja **-1**, esse será o valor de **pos** indicando que não encontramos o valor **n** no vetor **V**. Caso contrário, o valor de **pos** será exatamente a primeira posição onde o valor procurado foi encontrado no vetor V. Veja que ao final do programa temos uma estrutura de seleção para tratar essas situações e emitir a mensagem adequada a cada caso.

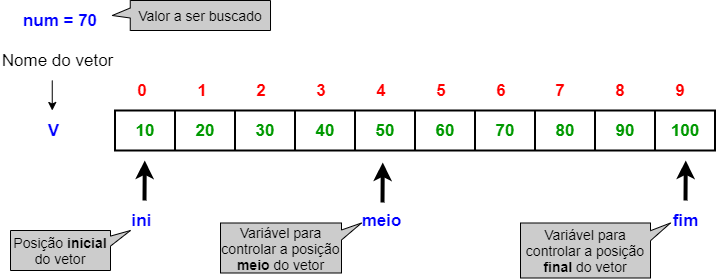
No lado direito da nossa figura 11.2 podemos verificar a função **busca**. Basicamente foi implementado um loop para fazer com que a variável **i** receba a cada passo do loop os valores das posições do vetor de 0 até a última posição que é **tam-1**. Dentro do loop usamos uma estrutura de seleção de forma que a cada passo do loop verificamos se o conteúdo de uma posição do vetor é igual ao valor que procuramos **V[i]==num.** Quando **i** é igual a 0 verificamos para a posição 0 do vetor e assim por diante. Caso o teste seja verdadeiro, encerramos a nossa busca através do comando **return i**. Lembrando que o comando return faz com que a função seja encerrada imediatamente. Nesse caso a função é encerrada e o valor **i** é retornado (o valor da posição onde o numero que estavamos procurando foi encontrado. Caso o valor não seja encontrado em nenhuma posição do vetor, então o loop é encerrado e retornamos **-1** indicando que não encontramos o valor. Notamos que nessa função temos dois comandos return mas apenas um deles é executado uma vez que esse comando encerra a função e não mais é executado.

Perceba que nesse algoritmo o melhor desempenho ocorre quando o elemento que buscamos aparece na primeira posição do vetor (1 teste). Já o pior desempenho ocorre quando o elemento não é encontrado e temos que testar todas as posições do vetor (n testes).

# 1.3. Busca Binária

Um outro método bastante eficiente de busca é a busca binária. Vamos verificar que esse método precisa de um número bem inferior de comandos executados mesmo considerando-se o pior caso. Esse algoritmo tem como pré-requisito que o vetor onde a busca irá ocorrer deve estar previamente ordenado. Portanto temos que ter em mente que ele só poderá ser usado nessas condições.

A busca binária é um algoritmo clássico de busca. O algoritmo se aproveita do fato dos valores do vetor estarem ordenados. Vamos entender seu funcionamento através de um exemplo Figura 11.3:

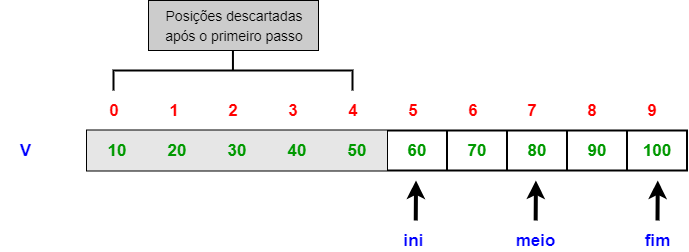


**Figura 11.3. Algoritmo de Busca Binária - Situação Inicial.**

**Fonte Autor.**

**Situação Inicial** - No início temos um vetor ordenado V, o número a ser buscado num, três variáveis indicando a posição de início, meio e fim do vetor. início começa com 0, fim com o valor da última posição do vetor 9 e meio com a média dessas posições (0+9)/2 = 4. Como as posições são números inteiros, desprezamos os valores após a vírgula. Figura 11.3

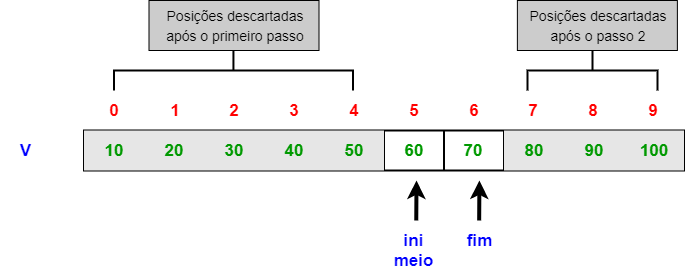
**Passo 1** - Verificamos se o valor que buscamos está na posição do meio (50). Como não está, descartamos a metade onde não está o número. Nesse passo como vetor está ordenado e o valor que se encontra no meio é 50, podemos descartar o lado esquerdo desse vetor pois temos certeza que o número buscado 70 não está desse lado. Fazemos então o acerto das variáveis **inicio**, **meio** e **fim** para: **inicio** = 5, **fim** = 9 e **meio** = (5+9)/2 = 7 Figura 11.4. Perceba que apenas nesse passo descartamos metade dos elementos do vetor. Vamos seguir para o passo 2.



**Figura 11.4. Algoritmo de Busca Binária - Passo 1.**

**Fonte Autor.**

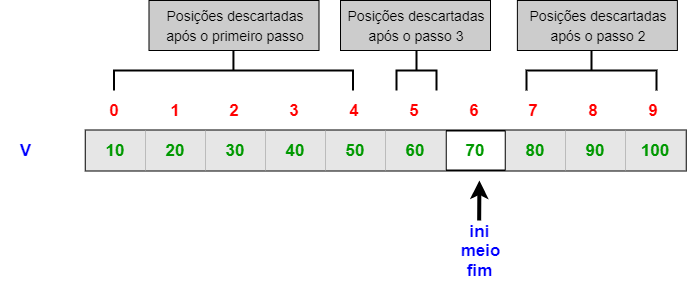
**Passo 2** - Verificamos se o valor que buscamos está na posição do meio (80). Como não está, descartamos a metade onde não está o número. Efetuamos o descarte de maneira semelhante ao passo 1. Descartando a metade onde não está o número considerando que o início agora é a posição 5. Como o valor que se encontra no meio é 80, podemos descartar o lado direito desse vetor pois temos certeza que o número buscado 70 não está desse lado (é menor que 80). Fazemos então o acerto das variáveis **inicio**, **meio** e **fim** para: **inicio** = 5, **fim** = 6 e **meio** = (5+6)/2 = 5 Figura 11.5.Vamos seguir para o passo 3.



**Figura 11.5. Algoritmo de Busca Binária - Passo 2.**

**Fonte Autor.**

**Passo 3** - Verificamos se o valor que buscamos está na posição do meio (60). Como não está, descartamos a metade onde não está o número.. Como o valor que se encontra no meio é 60, podemos descartar o lado esquerdo desse vetor pois temos certeza que o número buscado 70 não está desse lado (é maior que 60). Fazemos então o acerto das variáveis **inicio**, **meio** e **fim** para: **inicio** = 6, **fim** = 6 e **meio** = (6+6)/2 = 6 Figura 11.6. Vamos seguir para o passo 4.



**Figura 11.6. Algoritmo de Busca Binária - Passo 3.**

**Fonte Autor.**

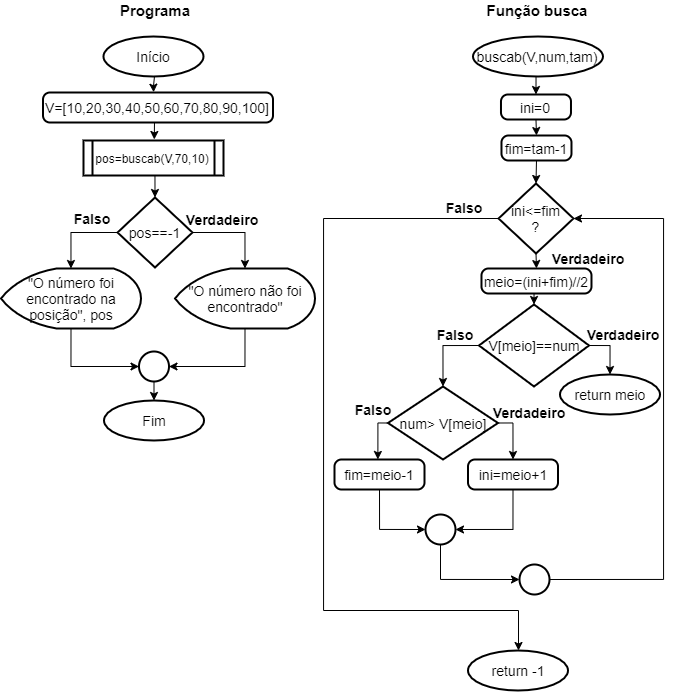
**Passo 4** - Verificamos se o valor que buscamos está na posição do meio (70). Como é o valor que buscamos, encerramos a busca retornando a posição onde foi encontrado (6). Como sobrou apenas 1 elemento, caso esse não fosse o valor encontrado retornaríamos -1 indicando que o valor não foi encontrado. Perceba que quando inicio=meio=fim significa que temos apenas um elemento,

Perceba então que nesse algoritmo a cada passo o número de posições que temos que procurar se reduz à metade. Pode-se provar que o número de buscas em função do tamanho do vetor **tam** pode ser calculado como log2 tam. Caso tam = 1024, =10, ou seja para um vetor de 1024 posições a busca precisa de no máximo em 10 passos. Comparando com a busca linear, o mesmo vetor no pior caso demoraria 1024 passos ou elementos verificados (o tamanho do vetor). É portanto 100 vezes mais rápido esse algoritmo para um vetor de 1024 posições.

Vamos verificar esse algoritmo usando fluxogramas:

**Exemplo 2** - Criar uma função que receba como parâmetro um vetor com números e um número a ser procurado. A função deve retornar a posição do vetor onde o número foi encontrado ou -1 caso não encontre o número. Usar o algoritmo de busca binária e considere que o vetor está ordenado em ordem crescente.

Podemos observar na Figura 11.7, do lado esquerdo temos a criação do vetor de 10 posições e a chamada da função **pos=buscab(V,70,10)**. Na chamada estamos passando o vetor **V**, o número a ser pesquisado **70** e o tamanho do vetor **10**. Em seguida temos o mesmo tratamento que foi feito na busca linear.Do lado direito temos a função que executa a busca binária. Na parte inicial da função armazenamos os valores iniciais das variáveis **inicio** e **fim** (primeira e última posição do vetor). Monta-se em seguida um loop que finaliza quando **inicio < fim** (a medida em que o algoritmo é executado o valor dessas variáveis vai se aproximando). No loop efetuamos os comandos vistos acima para cada passo. Calcula-se a posição do meio **meio=(ini+fim)//2**, verifica-se se o valor é encontrado nessa posição (V[**meio]==num**). Se for encontrado, encerramos o algoritmo e retornamos a posição **meio** através do comando **return meio**. Caso contrário, ajustamos as variáveis **inicio** e **fim**. Caso o valor seja maior do que o valor que está na posição do meio **num> v[meio]** desprezamos o lado esquerdo do vetor. Para isso ajustamos a variável **inicio=meio +1.** Caso contrário ajustamos desprezamos o lado direito ajustando a variável **fim=meio-1**. Se o valor for encontrado a função é encerrada. Caso o loop chegue ao final significa que o valor não foi encontrado e o comando **return -1** é executado, retornando o valor -1 o que significa que o valor buscado não foi encontrado.



**Figura 11.7. Fluxograma da busca binária com programa e função.**

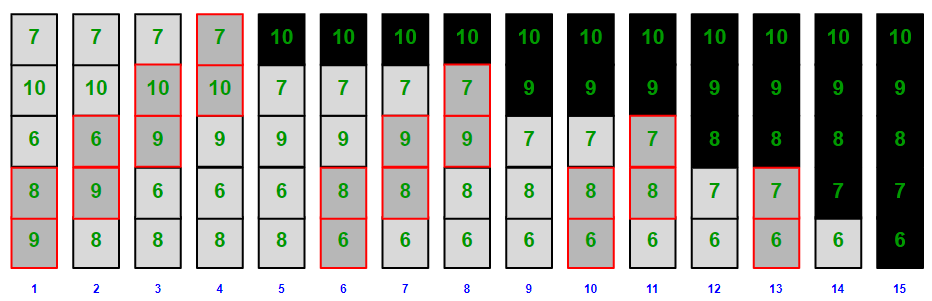
**Fonte Autor.**

# 1.4. Um algoritmo de ordenação Bubble Sort

Como vimos informações ordenadas facilitam e muito as buscas. Um exemplo disso é a busca de uma palavra no dicionário. O fato do dicionário estar ordenado agiliza é muito a nossa busca pelas cerca de 300.000 palavras existentes,

Vamos entender o funcionamento do algoritmo Bubble Sort ou método da bolha. Ele é chamado assim porque a medida em que o algoritmo é executado o valor ordenado vai subindo como bolhas. Veja na Figura 11.8 a sequência de passos na ordenação de um vetor. Ele foi colocado na vertical para sugerir exatamente o comportamento das bolhas. Seu funcionamento é o seguinte:

1. Fazemos a comparação entre duas posições vizinhas. Caso a posição anterior seja maior que a seguinte, efetuamos a troca dos valores dessas posições. Veja no passo 1 da Figura 11.8 efetuamos a comparação entre os valores 9 e 8. Como o 9 é menor que 8 efetuamos a troca (veja no passo 2). Essa troca será feita pela função **troca(v,p1,p2)** Figura 11.9 abaixo.
2. No passo 2 efetuamos a troca das 2 próximas posições 9 e 6. Como o 9 é maior do que o 6 efetuamos a troca desses valores.
3. E assim sucessivamente passos 3, 4 e 5 até a última posição do vetor. Perceba que sempre o maior valor de todos termina na posição mais acima. Veja que o valor 10 em preto ficou na última posição. Como ele é o maior, já está na sua posição final. Portanto ele não fará parte das próximas iterações. A cada iteração é chamada a função **empurra(v,n)** Figura 11.9 abaixo.
4. Após o passo 5, já percorremos todas as posições do vetor efetuando as comparações entre as posições vizinhas e já movimentamos o maior valor para a sua posição final.Vamos agora iniciar uma nova iteração que vai varrer novamente as posições do vetor menos a última pois já está na sua posição correta. Executamos então os passos 6,7,8 e 9 onde posicionamos o 9 que é o maior valor na sua posição correta. Perceba que a medida que o número fica na sua posição correta estamos colorindo a posição de preto.
5. E assim é feito sucessivamente até que o vetor esteja ordenado. Perceba que a cada iteração um número fica na posição correta e deixamos de percorrer uma posição, ou seja, na primeira iteração percorremos 5 posições, depois 4, 3 até 1.



**Figura 11.8. Fluxograma da busca binária com programa e função.**

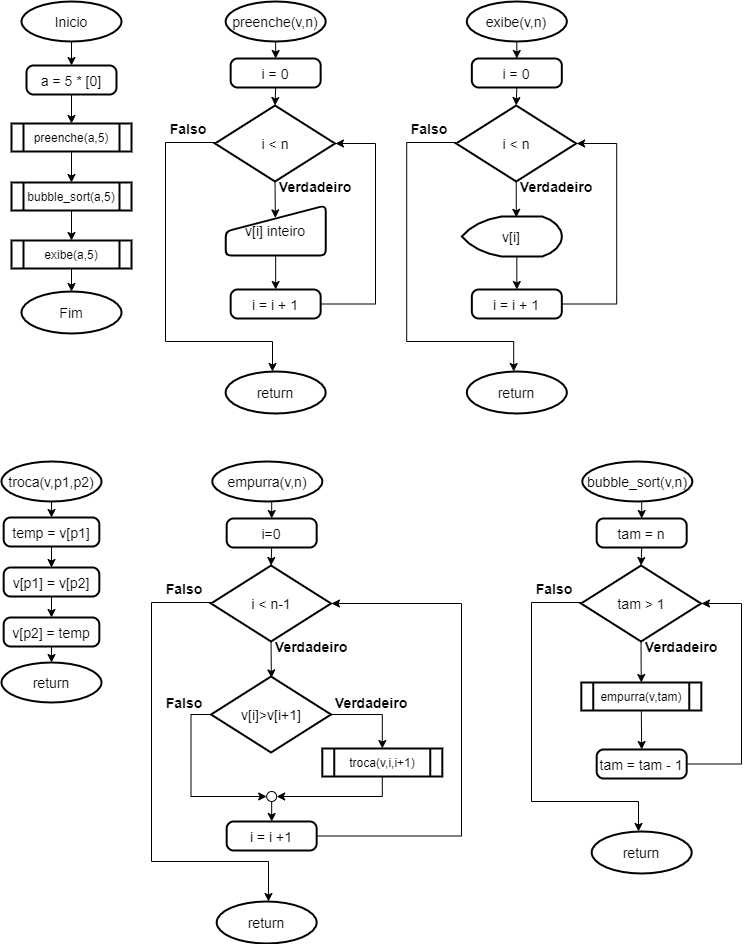
Para implementar o algoritmo vamos dividi-los em várias funções:

**Programa, funções preenche e exibe:** O programa cria o vetor, chama a função **preenche(a,5)** que preenche os valores do vetor, chama a função de ordenação usando o vetor preenchido **bubble(a,5)** e depois chama a função **exibe(a,5)** que vai exibir o vetor após a execução da função de ordenação. Espera-se que o vetor seja exibido ordenado. Todas essas funções possuem um segundo parâmetro com o tamanho do vetor.

**troca(v,p1,p2):** Essa função vai efetuar a troca entre duas posições **p1** e **p2** do vetor **v**. Após a execução dessa função o valor de v[p1] estará trocado com o valor de v[p2]

**empurra(v,n):** Executa o loop até o final do vetor (apenas 1 vez) verificando as posições vizinhas v[i]>v[i+1] e trocando (utilizando a função troca) caso o valor da posição anterior v[i] seja maior do que o valor da posição seguinte v[i+1]. A cada chamada da função empurra, um valor é ordenado. O objetivo dessa função é empurrar o maior valor para o final o vetor.

**bubble\_sort(v,n):** Essa função ao final de sua execução deixa o vetor v ordenado. Ela chama a função empurra n-1 vezes. A cada vez que chamamos a função empurra, o parâmetro de tamanho do vetor é reduzido, pois a cada vez que a função empurra é invocada, um elemento acaba ficando em sua posição final ordenada no final do vetor.

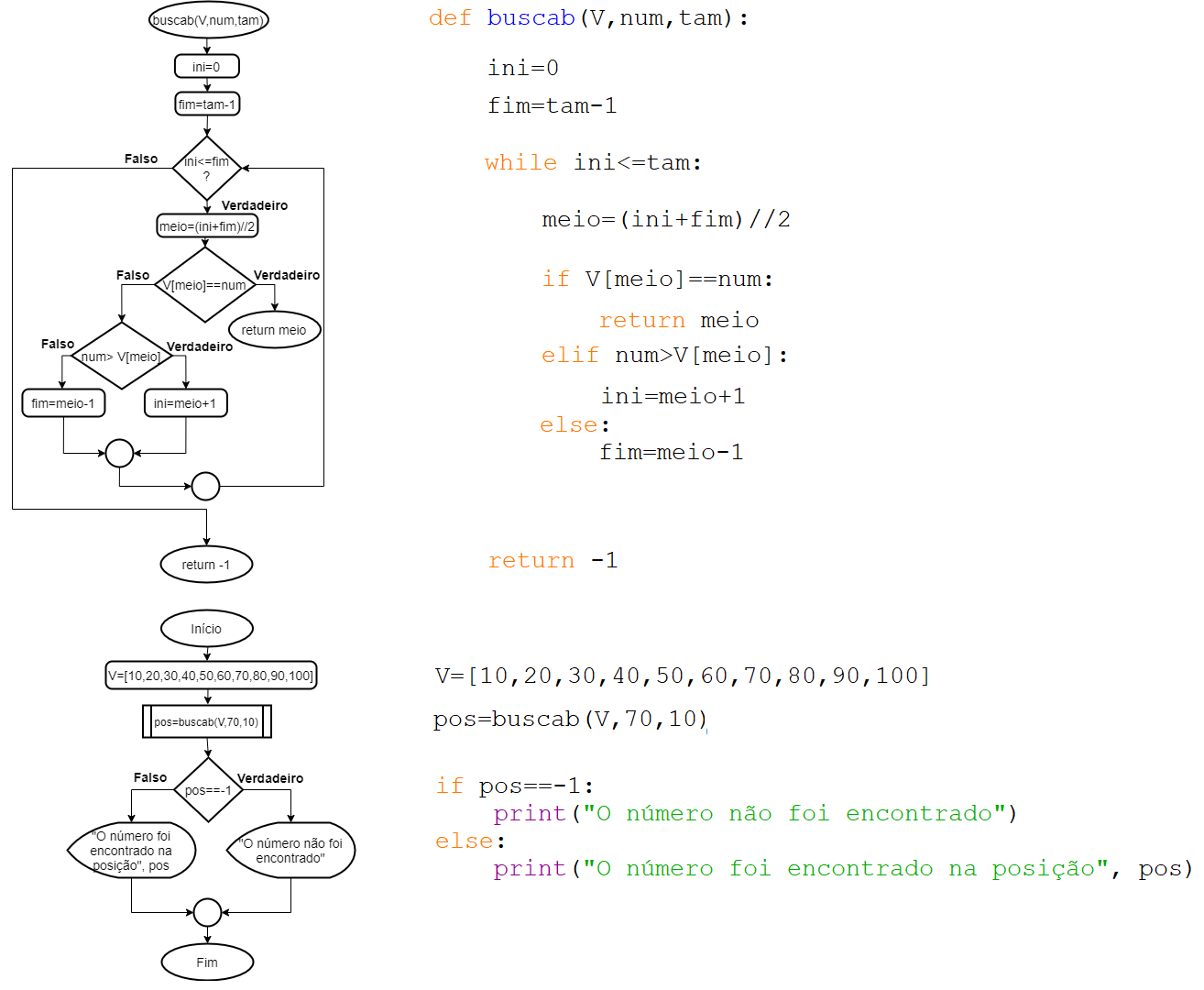


**Figura 11.9. Fluxograma com as funções usadas no algoritmo Bubble Sort.**

**1.5. Comparativo entre fluxograma e Python**

Abaixo podemos identificar um fluxograma da função do **exemplo 2** busca binária e sua conversão para python.

| **Fluxograma** | **Python** |
| --- | --- |

**Figura 10.17. Fluxograma e seu respectivo código em Python. Fonte Autor.**

**1.8. Referências**

DIERBACH, C. “Introduction to Computer Science Using Python: A Computational Problem-Solving Focus”1st Edition, New York: Wiley, 2012.

Python Tutor - <http://www.pythontutor.com/visualize.html>

# Quantas palavras existem na Língua Portuguesa? <https://www.vortexmag.net/quantas-palavras-existem-na-lingua-portuguesa>