Algoritmos de ordenação

Algoritmos de ordenação (ou *Sorting Algorithms*) podem ser definidos como:

Resumidamente, é o tipo de algoritmo que tem por entrada um *array* e organiza os seus itens seguindo uma ordem. Um fator a se considerar é que, as dimensões do *array*, e as limitações de espaço, podem influenciar na escolha do algoritmo que será utilizado.

Bubble sort – Tipo de Bolha

O *Bubble sort* é a escolha mais comum quando começamos a estudar sobre algoritmos e ordenação, por sua simplicidade e por dar uma ideia geral sobre como funcionam tais algoritmos.

O **bubble sort** realiza múltiplas passagem por uma lista. Ele compara itens adjacentes e troca aqueles que estão fora de ordem. Cada passagem pela lista coloca o próximo maior valor na sua posição correta. Em essência, cada item se desloca como uma "bolha" para a posição à qual pertence

Os passos são os seguintes:

1. Compare `A[0]` e `A[1]`. Se `A[0]` for maior que `A[1]`, troque os elementos;
2. Vá para `A[1]`. Se `A[1]` for maior que `A[2]`, troque os elementos; Repita o processo para cada par de elementos até o final do *array*;
3. Repita os passos `1` e `2` `n` vezes.

Abaixo o algoritmo representado em código *Python*:

```
def bubble_sort(array):
    """
    Teste de mesa
    -------------

    array = []:
```

```
        - `i` é `-1`, não cai no laço `while` e retorna array
sem modificações

    array = [1]:
        - `i` é `0`, não cai no laço `while` e retorna array sem
modificações

    array = [1, 2]:
        - `i` é `1`, entra no laço `while`
        - `range(1)` resulta em `[0]`
        - `j` é `0`. `j[0]` é menor que `j[1]`. Não faz swap
        - Sai do laço `for`
        - `i` é `0`. Sai do laço `while`
        - Retorna array sem modificações

    array = [3, 1, 2, 4]:
        - `i` é `3`, entra no laço `while`
        - `range(3)` resulta em `[0, 1, 2]`
        - `j` é `0`. `j[0]` é maior que `j[1]`. Faz swap
        - `array` fica [1, 3, 2, 4]
        - `j` é `1`. `j[1]` é maior que `j[2]`. Faz swap
        - `array` fica [1, 2, 3, 4]
        - `j` é `2`. `j[2]` é menor que `j[3]`. Não faz swap
        - Sai do laço `for`
        - `i` é `2`, entra no laço `for [0, 1]`. Itens estão
ordenados. Não faz swap
        - Sai do laço `for`
        - `i` é `1`, entra no laço `for [0]`. Itens estão
ordenados. Não faz swap
        - `i` é `0`. Sai do laço `while`
        - Retorna array ordenado
    """
    def swap(i, j):
        array[i], array[j] = array[j], array[i]

    i = len(array) - 1
    while i >= 0:
        for j in range(i):
            if array[j] > array[j + 1]:
                swap(i, j)
        i -= 1

    # Ao alterar o array in-place, não há reais motivos para
retorná-lo
    return array
```

- *Complexidade do tempo de execução*: Para cada elemento do *array*, o algoritmo faz `n - 1` comparações.

Considerando que ele percorre todo o array fazendo `n - 1` comparações, em *big O notation* temos `O(n^2)`.
- *Complexidade do espaço*: Como operamos a troca de elementos (`swap`), não precisamos de nenhuma outra estrutura de dados para armazenar o resultado da operação, com isso, temos complexidade de espaço de `O(1)`.

Selection sort

Outro algoritmo simples e intuitivo, e ligeiramente mais performático, é o *Selection sort*.

A ordenação por seleção (do Inglês, selection sort) é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os n-1 elementos restantes, até os últimos dois elementos.

Os passos podem ser resumidos em:

1. Encontre o menor valor;
2. Troque o menor valor com o primeiro item do *array*;
3. Encontre o segundo menor valor;
4. Troque o segundo menor valor com o segundo item do *array*;
5. Repita a operação até o *array* estar ordenado.

No algoritmo anterior tínhamos como finalidade mover o maior valor para o final do *array*. Nesse, a finalidade é selecionar o valor mais baixo no *array*, e movê-lo para o começo da estrutura.

```
def selection_sort(array):
    """
    Teste de mesa
    -------------
```

```
    array = []:
        - `len(array)` é `0`, não cai no primeiro no laço `for`
e retorna array sem modificações

    array = [1]:
        - `len(array)` é `1`, entra no primeiro laço `for`
        - `min_index` é `0`, não entra no segundo laço
        - Retorna array sem modificações

    array = [1, 2]:
        - `len(array)` é `2`, entra no primeiro laço `for`
        - `i` é `0` e `min_index` é `0`. Entra no segundo laço
`for`
        - `j` é `1`. `array[0]` não é maior que `array[1]`
        - Sai do segundo laço `for`
        - Faz swap `0` (`i`) e `0` (`min_index`)
        - Retorna array sem modificações

    array = [3, 1, 2, 4]:
        - `len(array)` é `4`, entra no primeiro laço `for`
        - `i` é `0` e `min_index` é `0`. Entra no segundo laço
`for`
        - `j` é `1`. `array[0]` é maior que `array[1]`
        - `min_index` é `1`
        - `j` é `2`. `array[1]` não é maior que `array[2]`
        - `j` é `3`. `array[1]` não é maior que `array[3]`
        - Sai do segundo laço `for`
        - Faz swap `0` (`i`) e `1` (`min_index`)
        - `array` fica [1, 3, 2, 4]
        - `i` é `1` e `min_index` é `1`. Entra no segundo laço
`for`
        - `j` é `2`. `array[1]` é maior que `array[2]`
        - `min_index` é `2`
        - `j` é `3`. `array[2]` não é maior que `array[3]`
        - Sai do segundo laço `for`
        - Faz swap `1` (`i`) e `2` (`min_index`)
        - `array` fica [1, 2, 3, 4]
        - `i` é `2` e `min_index` é `2`. Entra no segundo laço
`for`
        - `j` é `3`. `array[2]` não é maior que `array[3]`
        - Sai do segundo laço `for`
        - Faz swap `2` (`i`) e `2` (`min_index`)
        - `i` é `3` e `min_index` é `3`. Não entra no segundo
laço `for`
        - Faz swap `3` (`i`) e `3` (`min_index`)
        - Sai do primeiro laço `for`
        - Retorna array ordenado
    """
```

```
    def swap(i, j):
        array[i], array[j] = array[j], array[i

    for i in range(len(array)):
        min_index = i
        for j in range(i + 1, len(array)):
            if array[min_index] > array[j]:
                min_index = j

        swap(i, min_index)

    # Ao alterar o array in-place, não há reais motivos para
retorná-lo
    return array
```

- *Complexidade do tempo de execução*: Ao percebermos os dois *loops* alinhados, percorrendo a dimensão do *array* cada, podemos concluir que a complexidade é de `O(n^2)`.
- *Complexidade do espaço*: Como operamos a troca de elementos (`swap`), não precisamos de nenhuma outra estrutura de dados para armazenar o resultado da operação, com isso, temos complexidade de espaço de `O(1)`.

Insertion sort

Embora ele se compare aos dois algoritmos acima em *running time*, na minha opinião, imaginar o seu funcionamento demanda um pouquinho mais de esforço. Nesse algoritmo, o propósito é achar o lugar onde o elemento deveria estar no *array*:

1. Para cada elemento `A[i]`;
2. Se `A[i] > A[i + 1]`;
3. Troque os elementos até `A[i] <= A[i + 1]`.

Abaixo a implementação em *Python*:

```
def insertion_sort(array):
    """
    Teste de mesa
    -------------

    array = []:
        - `len(array)` é `0`, não cai no laço `for` e retorno
array sem modificações
```

```
    array = [1]:
        - `len(array)` é `1`, mas `range(1, len(array))` resulta
em `[]`
        - Retorna array sem modificações

    array = [1, 2]:
        - `len(array)` é `2`, entra no laço `for`
        - `slot` é `1`. `value` é `2` e `test_slot` é `0`
        - `test_slot` é maior que `-1`, mas `array[0]` não é
maior que `2`
        - `array[1]` recebe `2` (mesmo valor)
        - Retorna array sem modificações

    array = [3, 1, 2, 4]:
        - `len(array)` é `4`, entra no laço `for`
        - `slot` é `1`, `value` é `1` e `test_slot` é `0`
        - `test_slot` é maior que `-1` e `array[0]` é maior que
`1`. Entra no laço `while`
        - `array[1]` recebe `array[0]`
        - `test_slot` é `-1`
        - `array` fica [3, 3, 2, 4]
        - `test_slot` não é maior que `-1`. Sai do laço `while`
        - `array[0]` recebe `1`
        - `array` fica [1, 3, 2, 4]
        - `slot` é `2`, `value` é `2` e `test_slot` é `1`
        - `test_slot` é maior que `-1` e `array[1]` é maior que
`2`. Entra no laço `while`
        - `array[2]` recebe `3`
        - `test_slot` é `0`
        - `array` fica [1, 3, 3, 4]
        - `test_slot` é maior que `-1`, mas `array[0]` não é
maior que `2`. Sai do laço `while`
        - `array[1]` recebe `2`
        - `array` fica [1, 2, 3, 4]
        - `slot` é `3`, `value` é `4` e `test_slot` é `2`
        - `test_slot` é maior que `-1`, mas `array[2]` não é
maior que `4`. Não entra no laço `while`
        - `array[3]` recebe `4` (mesmo valor). Sai do laço `for`
        - Retorna array ordenado
    """

    for slot in range(1, len(array)):
        value = array[slot]
        test_slot = slot - 1

        while test_slot > -1 and array[test_slot] > value:
            array[test_slot + 1] = array[test_slot]
            test_slot = test_slot - 1
```

```
        array[test_slot + 1] = value

    # Ao alterar o array in-place, não há reais motivos para
retorná-lo
    return array
```

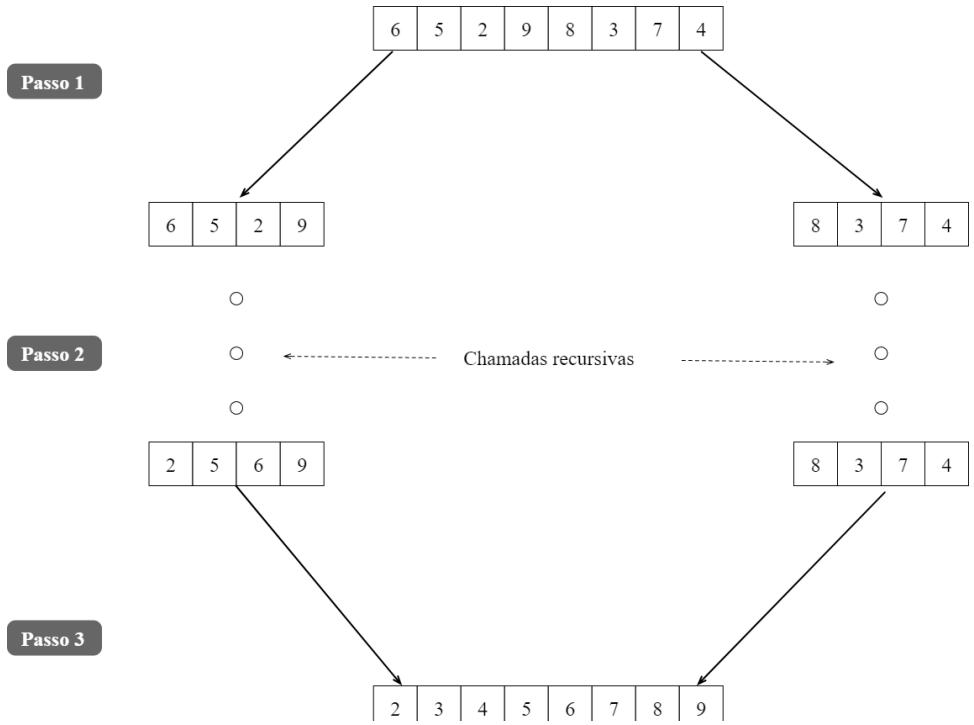
- *Running-time complexity*: No melhor cenário, o algoritmo terá complexidade `O(n)`. O mesmo apresenta complexidade `O(n^2)` como pior cenário.
- *Space complexity*: Como operamos a troca de elementos (`swap`), não precisamos de nenhuma outra estrutura de dados para armazenar o resultado da operação, com isso, temos complexidade de espaço de `O(1)`.

MergeSort

A ideia por trás do Mergesort é simples. Para um vetor A de n números, execute os seguintes passos:

1. Divida o vetor em duas metades.
2. Ordene recursivamente cada metade.
3. Mescle as duas metades do vetor, isto é, combine as duas metades para formar um único arranjo.

A figura abaixo ilustra o funcionamento do Mergesort.



Ao repetir o procedimento de divisão do vetor em duas metades, chegará um ponto em que teremos vários vetores de um elemento apenas. Por definição um arranjo de um elemento já está ordenado. Neste momento, precisamos fazer a junção (merge) dos vetores. Começamos combinando dois a dois todos os vetores de tamanho um, formando vetores ordenados de tamanho dois. O mesmo se repete para vetores de tamanhos dois, três, quatro, e assim por diante. Uma coisa que pode não parecer óbvia a

princípio (mas que ficará clara ao vermos a implementação do algoritmo) é que o trabalho pesado do *mergesort* é executado na etapa de merge, na qual dois sub-vetores ordenados são combinados em um único vetor ordenado.

Assim, a eficiência do *Mergesort* depende em quão eficientemente podemos combinar as duas metades ordenadas em um único arranjo ordenado. Podemos simplesmente concatenar as metades e então usar algum algoritmo de ordenação para ordenar o arranjo como um todo, ou então podemos combinar (mesclar, fazer o merge) das duas metades ordenadas em uma única sequência ordenada de forma eficiente. Mas como mesclar dois subvetores ordenados de forma eficiente? A ideia é mais ou menos a seguinte:

1. Seja k o tamanho de cada um dos subvetores V1 e V2.
2. Crie um vetor auxiliar de tamanho 2k.
3. Percorra os subvetores, sempre copiando para o vetor auxiliar o menor elemento na posição corrente dos subvetores, só avançando a posição no subvetor que teve o elemento copiado.

Com as ideias acima em mente, podemos implementar o *Mergesort* da seguinte forma:

```
import random
def merge(A, aux, esquerda, meio, direita):
    """
    Combina dois vetores ordenados em um único vetor (também ordenado).
    """
    for k in range(esquerda, direita + 1):
        aux[k] = A[k]
    i = esquerda
    j = meio + 1
    for k in range(esquerda, direita + 1):
        if i > meio:
            A[k] = aux[j]
            j += 1
        elif j > direita:
            A[k] = aux[i]
            i += 1
        elif aux[j] < aux[i]:
            A[k] = aux[j]
            j += 1
        else:
            A[k] = aux[i]
            i += 1
```

```
def mergesort(A, aux, esquerda, direita):
    if direita <= esquerda:
        return
    meio = (esquerda + direita) // 2 #posição i

    # Ordena a primeira metade do arranjo.
    mergesort(A, aux, esquerda, meio)

    # Ordena a segunda metade do arranjo.
    mergesort(A, aux, meio + 1, direita)

    # Combina as duas metades ordenadas anteriormente.
    merge(A, aux, esquerda, meio, direita)


# Testa o algoritmo.
A = random.sample(range(-10, 10), 10)
print("Arranjo não ordenado: ", A)
aux = [0] * len(A)
mergesort(A, aux, 0, len(A) - 1)
print("Arranjo ordenado:", A)
```

```
Arranjo não ordenado:  [0, 7, 8, 2, -2, 1, -5, 6, 3, -1]
Arranjo ordenado: [-5, -2, -1, 0, 1, 2, 3, 6, 7, 8]
```

Existem várias versões de implementações do *mergesort*.

No pior caso, o *Mergesort* possui complexidade $O(n \log n)$ no número de elementos do vetor de entrada, ele é um algoritmo estável, mas não é *in-place*, pois usa um vetor auxiliar para combinar os sub-vetores ordenados.

Note que a complexidade do *Mergesort* é inferior à dos algoritmos de ordenação por seleção e inserção. Na verdade, como veremos mais à frente, $O(n \log n)$ é a melhor complexidade que podemos obter para um algoritmo de ordenação genérico. O *Bucketsort* e o *Radixsort* possuem complexidade linear, mas não são algoritmos genéricos – eles só funcionam para entradas específicas.

Para entender por que o *Mergesort* possui complexidade $O(n \log n)$, precisamos analisar a complexidade dos dois procedimentos que o compõem:

- `mergesort`: simplesmente divide o vetor de entrada em duas metades e invoca o procedimento `merge`. A divisão do vetor de entrada em dois possui complexidade logarítmica, pois o vetor original é dividido em duas metades $\log n$ vezes. Por exemplo, se o vetor de entrada possui tamanho 64, ele será dividido em vetores de tamanhos 32, 16, 8, 4, 2, e 1, ou seja, ele será dividido $6 = \log 64$ vezes.
- `merge`: o procedimento `merge` possui complexidade $O(n)$. Entretanto, ele é executado $\log n$ vezes para um vetor de tamanho n. Assim, a complexidade final do *Mergesort* será $O(n \log n)$.

QuickSort

O QuickSort é um algoritmo de divisão e conquista. Ele seleciona um elemento como pivô e particiona o array fornecido ao redor do pivô selecionado. Existem muitas versões diferentes do quickSort que selecionam o pivô de maneiras diferentes.

1. Sempre escolha o primeiro elemento como pivô.
2. Sempre escolha o último elemento como pivô (implementado abaixo)
3. Escolha um elemento aleatório como pivô.
4. Escolha a mediana como pivô.

O principal processo no quickSort é a partição (). O destino das partições é, dado uma matriz e um elemento x da matriz como pivô, colocar x em sua posição correta na matriz classificada e colocar todos os elementos menores (menores que x) antes de x, e colocar todos os elementos maiores (maiores que x) depois x. Tudo isso deve ser feito em tempo linear.

O quicksort é um clássico dos algoritmos de ordenação, até mesmo fazendo parte das bibliotecas da linguagem C. Assim como o heapsort, ele também é estruturado como uma Árvore Binária. Sua estratégia de ordenação é baseada no padrão de projeto (Design Pattern) divisão e conquista, utilizando como parte da estratégia a recursividade.
A execução do quicksort consiste em dividir uma sequência em subsequências menores recursivamente. Primeiro, é escolhido um elemento qualquer da sequência como referência, o qual é chamado de pivô (pivot). Geralmente escolhe-se o primeiro ou o último elemento da sequência, evitando o tempo de processamento da escolha aleatória. Em seguida, ordena-se os elementos em subsequências com os elementos

menores à esquerda (left) e os elementos maiores a direita (right) através da recursão e, finalmente, concatenando as subsequências ordenadas até obter a lista ordenada.
O algoritmo quicksort consome tempo proporcional a O(n log n) em média, em geral é muito rápido. Porém, em algumas raras instâncias ele pode ser tão lento quanto os algoritmos elementares, com tempo de execução de $O(n^2)$ no pior caso. Por esta razão não é recomendado utilizar o quicksort em projetos de tempo real. É possível eliminar essa situação de pior caso $O(n^2)$ com a versão randomizada do quicksort. A versão randomizada utiliza um pivô aleatório ao invés de utilizar um índice fixo do início ou final do arranjo, com isso obtendo na ordem de pior caso O(n long n).

Heapsort 19-10

A cada iteração do *Heapsort*, selecionamos o elemento mínimo e o colocamos na posição correta. Repetimos esse procedimento para todos os elementos do vetor. Entretanto, em vez de usarmos um arranjo simples, usamos um heap, que nos permite encontrar o elemento mínimo a um custo computacional mais baixo. O custo de se encontrar o menor elemento em um heap é O(1), mas a cada vez que extraímos o menor elemento temos que atualizar o heap – complexidade O(log n). Como fazemos isso para os n elementos do vetor de entrada, a complexidade assintótica do *Heapsort* é O(n log n). A ideia é a mesma do algoritmo de ordenação por seleção, mas por causa da estrutura de dados usada (heap em vez de arranjo simples) a complexidade assintótica acaba sendo bem melhor. O *Heapsort* é um exemplo de como o uso da estrutura de dados correta pode fazer a diferença entre um algoritmo eficiente e um algoritmo ineficiente.

O concorrente mais direto do quicksort é o heapsort. O Heapsort geralmente é um pouco mais lento que o quicksort, mas o pior caso de execução é sempre Θ (nlogn). O Quicksort geralmente é mais rápido, embora ainda exista a chance de pior desempenho, exceto na variante introsort, que muda para heapsort quando um caso ruim é detectado. Se for sabido antecipadamente que o heapsort será necessário, usá-lo diretamente será mais rápido do que esperar que o introsort mude para ele.

O Heapsort é O (N log N) garantido, o que é muito melhor do que o pior caso no Quicksort. O Heapsort não precisa de mais memória para outra matriz para colocar dados ordenados conforme necessário pelo Mergesort. Então, por que os aplicativos comerciais ficam com o

Quicksort? O que o Quicksort tem que é tão especial em relação a outras implementações?

O segredo do Quicksort é: ele quase não faz trocas desnecessárias de elementos. A troca é demorada.

Com o Heapsort, mesmo se todos os seus dados já tiverem sido solicitados, você trocará 100% dos elementos para solicitar a matriz.

Com o Mergesort, é ainda pior. Você gravará 100% dos elementos em outra matriz e os gravará novamente na original, mesmo que os dados já estejam ordenados.

Com o Quicksort, você não troca o que já está pedido. Se seus dados forem completamente ordenados, você não trocará quase nada! Embora exista muita discussão sobre o pior caso, uma pequena melhoria na escolha do pivô, além de obter o primeiro ou o último elemento da matriz, pode evitá-lo. Se você obtiver um pivô do elemento intermediário entre o primeiro, o último e o meio, será suficiente para evitar o pior caso.

O que é superior no Quicksort não é o pior caso, mas o melhor! Na melhor das hipóteses, você faz o mesmo número de comparações, ok, mas você troca quase nada. Em geral, você troca parte dos elementos, mas não todos, como no Heapsort e Mergesort. É isso que dá ao Quicksort o melhor momento. Menos troca, mais velocidade.

Algoritmos de Busca de Palavras em Texto

A busca de padrões dentro de um conjunto de informações tem uma grande aplicação em computação.

São muitas as variações deste problema, desde procurar determinadas palavras ou sentenças em um texto até procurar um determinado objeto dentro de uma sequência de bits que representam uma imagem.

Todos eles se resumem a procurar certa sequência de bits ou bytes dentro de uma sequência maior de bits ou bytes.

Vamos considerar a versão de procurar uma sequência de bytes dentro de outra sequencia, ou ainda, procurar uma **palavra** dentro de um texto. **Palavra** deve ser entendida como uma sequência qualquer de caracteres.

Usando a terminologia do Python, estamos considerando então procurar uma sub-string dentro de uma string de caracteres. Já existe uma função intrínseca com essa função em Python (find). Essa função devolve o índice da primeira ocorrência de uma sub-string dentro de outra, ou -1 se não encontrar. Exemplos:

```
str1 = "sentimento de aumento do cimento"
str2 = "men"

print(str1.find(str2))                # imprime 5

print(str1.find(str2, 25))       # imprime 27
print(str1.find(str2, 15, 25))   # imprime 16
```

Estamos interessados em saber quantas vezes uma sub-string ocorre dentro de uma string. Tal função também já existe em Python (count). Exemplos:

```
print(str1.count(str2))             # imprime 3
print(str1.count(str2, 25))         # imprime 1
print(str1.count(str2, 15, 25))     # imprime 1
print(str1.count("e"))              # imprime 5
```

O nosso objetivo é saber como construir esse algoritmo. A formulação do problema fica então:

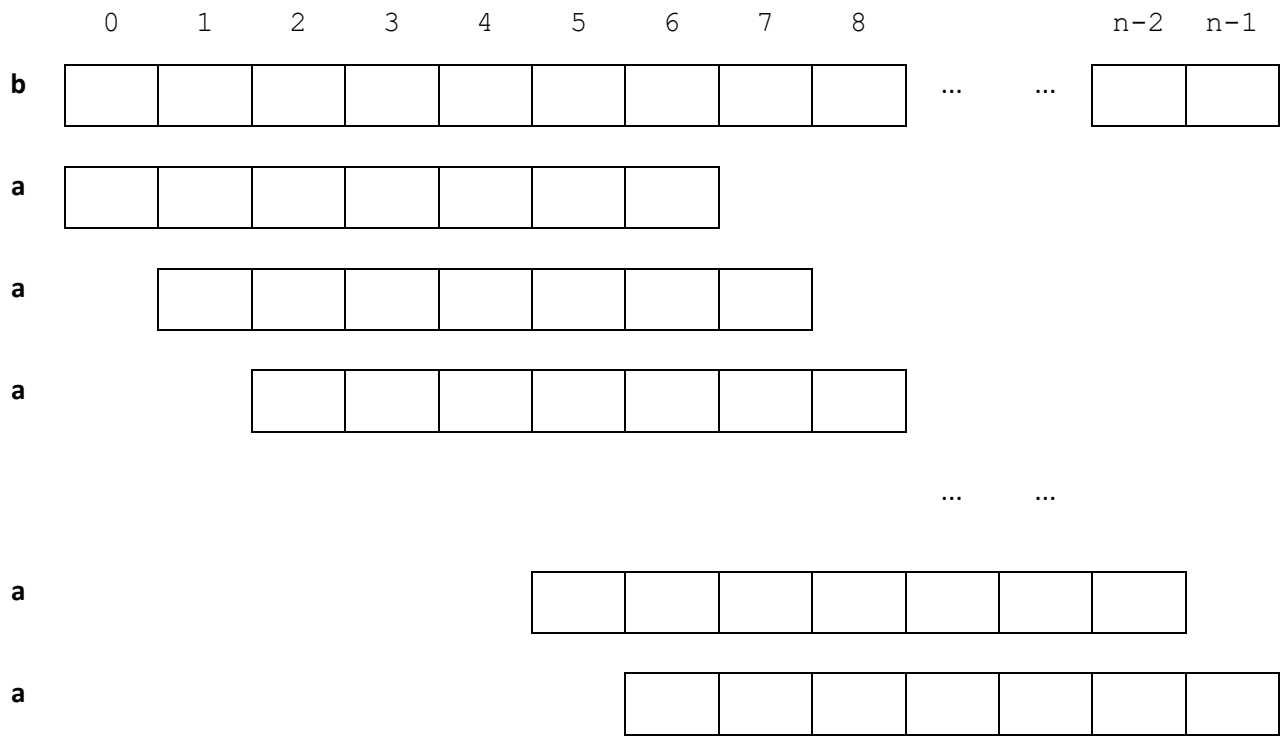
Dada uma sequência `a` de `m > 0` bytes (`a[0], ..., a[m - 1]` ou `a[0:m]`) **verificar quantas vezes** ela ocorre em uma sequência `b` de `n` elementos (`b[0], ..., b[n - 1]` ou `b[0:n]`).

Usando a notação do Python – Verificar **quantas vezes** a string **`a[0:m]`** ocorre em **`b[0:n]`**.

O algoritmo tradicional

A solução mais trivial deste problema consiste então em comparar:

```
a[0] com b[0]; a[1] com b[1]; ... a[m - 1] com b[m - 1]
a[0] com b[1]; a[1] com b[2]; ... a[m - 1] com b[m]
...
a[0] com b[n-m]; a[1] com b[n-m+1]; ... a[m-1] com b[n-1]
```



Na primeira comparação em que **a[i]** diferente de **b[j]**, passa-se para o próximo passo.

Exemplos:

b - O alinhamento do pensamento provoca casamento

a – mento – Ocorre 3 vezes

a – n – Ocorre 5 vezes

a – casa – Ocorre 1 vez

a – ovo – Ocorre 1 vez

a – prova – Ocorre 0 vezes

b – abababababa

a – bab – Ocorre 3 vezes

a – abab – Ocorre 3 vezes

a – bababa – Ocorre 2 vezes

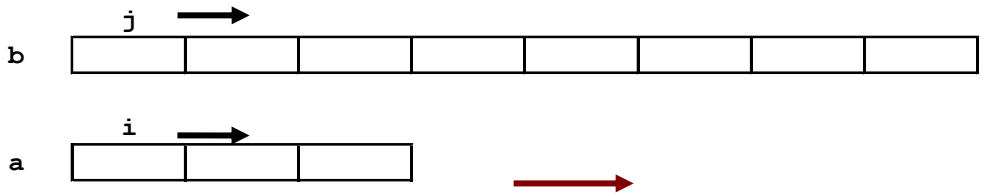
Abaixo esta primeira solução:

```
def comparapadrao(a, b):
    m, n = len(a), len(b)
    conta = 0
    for k in range(n - m + 1):
        i, j = 0, k
        while i < m:
            print(i, j)
            if a[i] != b[j]: break
            i, j = i + 1, j + 1
```

```
            if i == m: conta += 1
        return conta
```

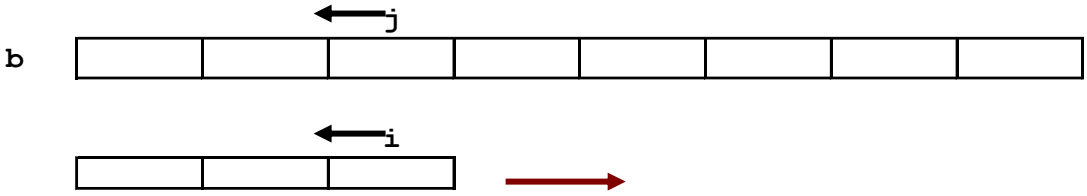
Na solução acima, `i` e `j` caminham da esquerda para a direita.

Também `a` se desloca da esquerda para a direita a cada nova tentativa.



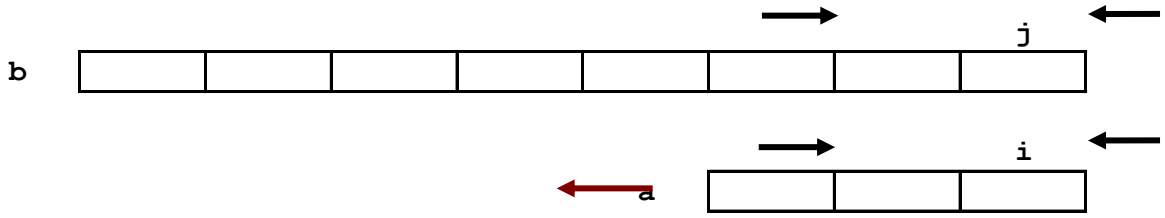
Podemos ter algumas variações, todas equivalentes:

Procurando `a` em `b` da direita para a esquerda:




```
def comparapadrao1(a, b):
    m, n = len(a), len(b)
    conta = 0
    for k in range(m - 1, n):
        i, j = m - 1, k
        while i >= 0:
            if a[i] != b[j]: break
            i, j = i - 1, j - 1
        if i < 0: conta += 1
    return conta
```

Varrendo b da direita para a esquerda e procurando a em b da esquerda para a direita ou da direita para a esquerda:



Exercícios:

1) Adapte o algoritmo acima, varrendo b da direita para a esquerda e procurando a em b da esquerda para a direita.
2) Idem, varrendo b da direita para a esquerda e procurando a em b da direita para a esquerda.
3) Quantas comparações são feitas no mínimo e no máximo? Encontre sequências a e b onde o mínimo e o máximo ocorrem.
4) Por que a complexidade dos algoritmos acima é $O(n^2)$?
5) Explique porque as versões acima são todas equivalentes.
6) Adapte os algoritmos acima, devolvendo o índice inicial em b da primeira ocorrência de a ou -1 se não encontrar.