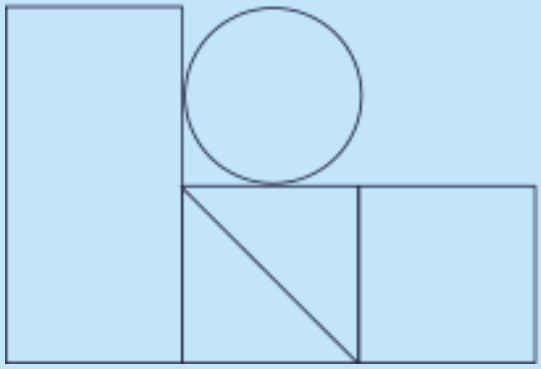
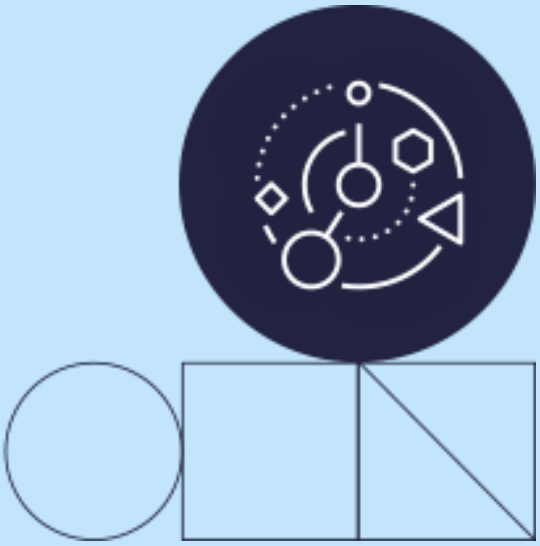
IBM SkillsBuild

IBM

Programação em Python

Algoritmos de busca linear e binária





Índice

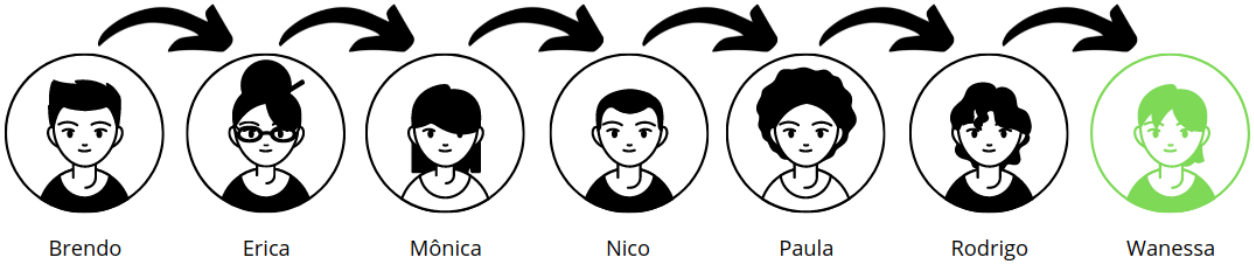
Introdução

Como vimos em tópicos anteriores, hoje em dia, a necessidade de algoritmos mais eficientes para nossas aplicações tornou-se mais visível, seja por causa da quantidade de dados processados, seja por causa da necessidade de respostas rápidas. Isto nos leva a um dos principais fundamentos do desenvolvimento de software: analisar a complexidade dos algoritmos.

Um exemplo prático: imaginemos que estamos desenvolvendo uma lista telefônica e somos responsáveis pela criação da função de busca de contatos, assumindo que todos os contatos já estejam em ordem alfabética.

Busca linear

Para facilitar e evitar erros, **vamos percorrer a lista** um a um procurando por um contato específico. Vamos supor que o contato de Wanessa foi solicitado:



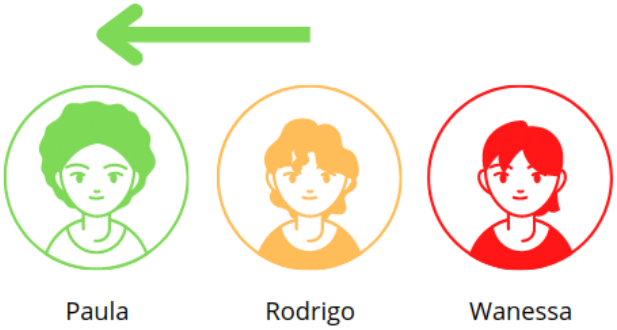
Descobrimos que percorremos toda a lista até encontrá-la. A solução atual funciona perfeitamente e com desempenho aceitável. Agora imagine que nossa solução foi vendida a uma empresa multinacional que tem milhares ou até milhões de contatos.

Desta vez, teremos que percorrer a lista de milhões de contatos e rolar pela lista até encontrarmos o contato solicitado. Como podemos ver, para este tipo de busca, a solução linear não parece ser a mais eficaz.

Busca binária

Vamos pensar em uma solução melhor. Sabemos que a lista já está em ordem alfabética, então podemos quebrá-la e comparar o contato que estamos procurando com o contato no meio, verificar a direção que precisamos seguir e eliminar o resto da lista. Portanto, seguiremos os passos descritos acima para encontrar o contato de Paula:

Selecionamos o contato do meio, Nico, para comparar com o contato que estamos procurando, Paula, e descartamos a metade da lista onde temos certeza de que a pessoa que estamos procurando não estará presente.

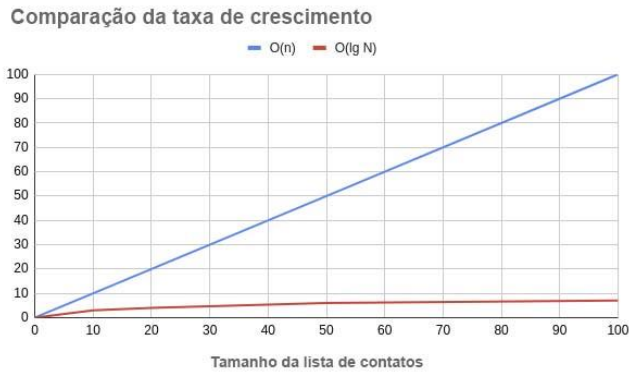


Nesta iteração, escolhemos novamente o contato no meio da lista, Rodrigo, e verificamos em que direção devemos ir, descartando a outra metade. Depois encontramos Paula.

Qual seria a diferença entre as duas soluções?

Note que na primeira solução, independentemente do tamanho da lista, temos que passar por todos os contatos até encontrarmos aquele que queremos. Se tivermos sorte, o contato que estamos procurando pode ser o primeiro de todos eles. Entretanto, se não tivermos tanta sorte e procurarmos o último contato, teremos que percorrer toda a lista, o que seria o pior cenário para o algoritmo. Com isso em mente, podemos determinar que nossa primeira solução executa uma função que cresce linearmente com o tamanho da lista de contatos.

Na segunda solução, a cada iteração retiramos metade da lista, de modo que não precisamos passar por toda a lista. Desta forma, otimizamos a busca. Esta solução realiza uma função que cresce a uma taxa logarítmica considerando o tamanho da lista de contatos.



Agora está claro que a solução logarítmica tem um desempenho extremamente mais eficiente do que a solução linear, e podemos confirmar isso pelo gráfico acima, onde o eixo do número de operações realizadas pelas duas funções é bastante desigual.

Indo um pouco mais fundo, em uma lista com 1 milhão de contatos, é necessário apenas realizar 20 operações de comparação até que o contato desejado seja encontrado. Com isso, podemos começar a entender a importância de analisar a complexidade dos algoritmos.

Como outro exemplo, imagine ter que desenvolver uma função de busca de estudantes para tornar possível o login em um web site. Após encontrar a pessoa, nossa intenção é autentificá-la para autorizar o acesso à plataforma. Para isso, recebemos uma lista com todos os nomes das pessoas registradas.

Em primeiro lugar, utilizaremos a busca linear. Então decidimos percorrer toda a lista procurando a pessoa a ser autenticada e chegamos à seguinte implementação:

```
from array import array

def buscar(lista, nombre_buscado):
    tamano_lista = len(lista)
    for actual in range(0, tamano_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos =
sorted(importa_lista('../data/lista_alumn
os'))
    posicion_del_alumno =
buscar(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

Desenvolvemos a função de busca com uma lista contendo 7 alunos e também simulamos a busca com aproximadamente 85.000 alunos. Tudo funcionou bem e com desempenho aceitável, como mostra o seguinte algoritmo:

```
def buscar(lista, nombre_buscado):
    tamano_lista = len(lista)
    for actual in range(0, tamano_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos =
sorted(importa_lista('../data/lista_alumn
os'))
    posicion_del_alumno =
buscar(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()




def importar_lista(archivo):
    lista = []
    with open(archivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
        lista.append(line)
    return lista
```

```
def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    for actual in range(0,
tamano_de_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos =
sorted(importa_lista('../data/lista_aluno
s'))
    posicao_do_aluno =
busca(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

Agora vamos assumir que o cliente confirma que receberemos aproximadamente 3000 logins de novas pessoas no dia seguinte.

Decidimos fazer uma simples simulação para ver como nossa busca reagiria a esta quantidade de solicitações. E, pensando no pior cenário, sempre optaremos pela busca da última pessoa da lista.

```
from array import array

def importar_lista(archivo):
    lista = []
    with open(arquivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
        lista.append(line)
    return lista

def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    for actual in range(0,
tamano_de_lista):
        if (lista[actual] ==
nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos = ["Brendo",
"Erica", "Monica", "Nico", "Paulo",
"Rodrigo", "Wanessa"]
    for i in range(0, 3500):
        posicion_del_alumno =
buscar(lista_de_alumnos, "Wanessa")
        print("Alumno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

Mudamos a chamada de busca e fizemos um loop para simular 3500 pedidos de busca. Na execução, notamos que o tempo aumentou significativamente.

Por que isso aconteceu? Vamos fazer um cálculo simples para responder a esta pergunta:

Para cada um dos 3500 pedidos de busca foram realizadas 85 mil comparações, ou seja: (3500*85000) = 297500000 operações.

E se a lista de pessoas ou pedidos crescer, o algoritmo crescerá linearmente até essas quantidades. Portanto, assumimos que cada um desses algoritmos é O(N).

Agora vamos otimizar o algoritmo.

Pensando nos estudantes que receberemos, vamos otimizar este algoritmo usando a técnica de dividir e conquistar usando a busca binária.

```
from array import array

def importar_lista(archivo):
    lista = []
    with open(arquivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
        lista.append(line)
    return lista

def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    inicio = 0
    fim=tamano_de_lista-1

    while inicio<=fin:
        medio=(inicio+fin)//2
        if lista[medio] ==
nombre_buscado:
            return medio
        elif lista[meio] <
nombre_buscado:
            inicio=medio+1
        elif lista[medio] >
nombre_buscado:
            fin = medio-1

    return -1

def main():
    lista_de_alumnos =
sorted(importar_lista('../data/lista_alum
nos'))
    for i in range(0,3500):
        posicion_del_alumno =
buscar(lista_de_alumnos, "Zoraida")
        print("Aluno(a) {} está en la
posicion
{}".format(lista_de_alumnos[posicion_del_
alumno], posicion_del_alumno))

if __name__ == "__main__":
    main()
```

Primeiro, usamos a função **sorted()** de Python para retornar a lista em ordem alfabética. A partir daí, começamos a procurar o estudante Zoraida, que está no final da lista, para simular o pior cenário possível.

Refatoramos a função de busca para usar a busca binária, que consiste em comparar o valor procurado com o valor do elemento no meio da lista e, se forem iguais, a posição do meio é devolvida.

```
if lista[medio] == nombre_buscado:
            return medio
```

Se o valor pesquisado precede o valor médio, o algoritmo descarta todos os valores subseqüentes.

```
elif lista[medio] > nombre_buscado:
            fim = medio-1
```

E se o valor pesquisado estiver após o valor médio, o algoritmo descarta todos os valores anteriores, até que apenas o elemento desejado permaneça. Se o elemento restante não for o que queremos, um valor negativo é devolvido.

```
elif lista[medio] < nombre_buscado:
            inicio=medio+1
```

Agora, quando simulamos as 3500 solicitações, percebemos que a execução é quase instantânea. Por que este algoritmo funciona tão rápido? Vamos analisá-lo.

Comparação da eficiência dos algoritmos

Considerando, mais uma vez, que temos 85.000 alunos e que em cada iteração de busca é descartada metade da lista que não é a que estamos procurando, podemos calcular usando logaritmos na base 2 e concluir que com cada solicitação da função de busca, no pior dos casos, são realizadas operações de **lg(N)**, ou seja,

lg(85000) ≃ 16 operações.

Mas ainda não acabou, pois fizemos 3500 chamadas para essa função, então fizemos **(3500 * 16) ≃ 56000** operações.

Assim, conseguimos otimizar nosso algoritmo, que anteriormente realizava quase 300 milhões de operações, para apenas 56 mil.

Mais uma vez provamos a importância de pensar sobre como nossos algoritmos se comportarão de acordo com a quantidade de dados recebidos, e vale a pena ressaltar que, apesar de haver abstrações de tais funções, como a função **index()** que faria esta busca com apenas uma chamada, é muito importante aprender estas noções básicas.

É importante notar que para a solução de busca binária é necessário ter a lista ordenada, por isso usamos a função **sorted()** para ordená-la.