Análise de Algoritmos: Introdução

A análise de grandes volumes de dados (*Big Data Analytics*) vem se tornando cada vez mais comum para instituições em diversos setores devido aos movimentos de Inteligência Empresarial (*Business Intelligence*), Indústria 4.0, veículos inteligentes, Internet das Coisas (IoT), redes de sensores, etc. que geram uma explosão de dados. Com isso, torna-se cada mais importante termos algoritmos eficientes. O ponto é que **podemos utilizar há diversos algoritmos possíveis para resolver o mesmo problema**. Então, a pergunta é: **como analisar a eficiência de um algoritmo**?

Nesta série de postagens, a análise da eficiência de algoritmos será apresentada. A eficiência de um algoritmo pode estar relacionada com o **tempo de execução** ou **espaço** utilizado pelo mesmo. O foco desta série será no **tempo de execução**.

Pré-requisitos

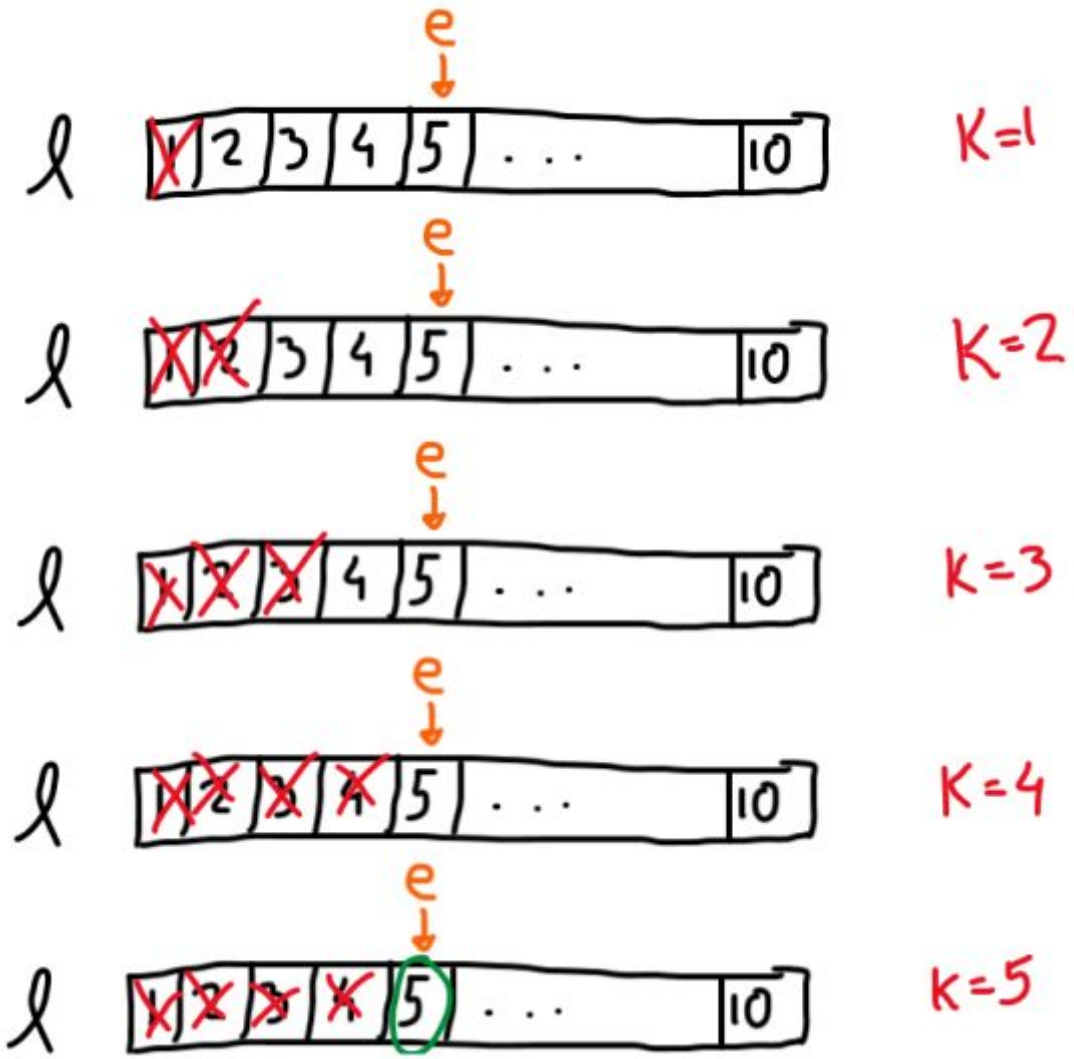
Para esta série, é esperado que você tenha domínio dos conceitos básicos de algoritmo, como variáveis e estruturas de condição e repetição. Os exemplos de código são apresentados em **Python**.

Calculando o tempo de execução

Uma abordagem muito simples e óbvia para calcular o tempo de execução de um algoritmo é **executar o algoritmo em questão** e cronometrar o tempo necessário para a execução do mesmo. Para exemplificar, vamos usar o exemplo de um jogo que todo mundo deve conhecer: adivinhação de número.

Nesse jogo, temos duas pessoas: João e Maria. João pensa em um número de 1 a 100. Maria deve tentar adivinhar qual o número que João pensou. Para tal, ela deve chutar um número, e João deve dizer **ALTO**, se o chute foi alto (ou seja, o número chutado é maior do que o pensado); **BAIXO**, se o chute foi baixo; e **CERTO**, se ela acertou.

Há várias abordagens que Maria pode usar para tentar adivinhar o número pensado por João. Uma delas, a mais simples, seria ela começar chutando 1 (o menor número) e ir incrementando as tentativas por 1 (chutar, na sequência, 1, 2, 3, 4, etc.) até acertar o número. Com esta abordagem, se João tivesse pensado no número 100 (**o pior caso**), Maria precisaria de 100 tentativas para acertar o número. Caso ele tivesse pensado no número 1 (**o melhor caso**), ela precisaria de apenas 1 tentativa. Esse algoritmo é denominado **busca linear**. Na Figura abaixo, eu ilustro a busca assumindo que João pensou no número 5, em uma lista de 1 a 10.



Na Figura acima, K representa o número de chutes até o momento. Pode-se observar que são necessários 5 chutes (comparações) para acertar (encontrar) o número 5 em uma **busca linear**considerando uma lista de 1 a 10.

O pseudocódigo da **busca linear** é apresentado abaixo:

```
1 proc buscalinear(lista l, elemento e)
2    k = 1
3    enquanto k for menor que o número de elementos na lista faca
4        se l[k] == e
5            retornar VERDADEIRO
6        caso contrário
7            k = k + 1
8    retornar FALSO
```

No contexto do nosso exemplo, para o pseudocódigo acima, as entradas **l** e **e** representam, respectivamente, uma lista contendo os números de 1 a 100 e o número pensado por João. O bloco entre as linhas 3 e 8 itera em todos os elementos da lista, representando a sequência de chutes que Maria vai tentar (1, 2, 3, 4 … 100). No caso do nosso exemplo, seriam os números de 1 a 100. A linha 4 compara o chute de Maria com o número pensado por João, no caso, isso seria João respondendo (BAIXO), pois estamos buscando de forma linear começando pelo menor número. Ou seja, quando Maria errar, ela sempre vai errar para baixo! Seria o inverso, caso Maria começasse a chutar de 100, e baixasse até 1.

A linha 5 apenas é executada se Maria acertar o número, retornando VERDADEIRO. Caso Maria tenha errado o chute, executa-se as linhas 6 e 7, e o valor do chute (k) é incrementado. Caso Maria nunca acerte o número (isso não vai ocorrer nunca para esse caso, pois Maria tentará todos os possíveis!), a linha 8 seria executada, retornando FALSO.

Para testar esse algoritmo, a gente precisa implementar o algoritmo em uma linguagem de programação e testar com algumas entradas (também chamadas de **instâncias do problemas**). O código em Python é apresentado abaixo (você pode modificar o código aqui). Eu fiz

algumas alterações no pseudocódigo original para que a gente possa analisar o tempo de execução do algoritmo para alguns casos de teste.

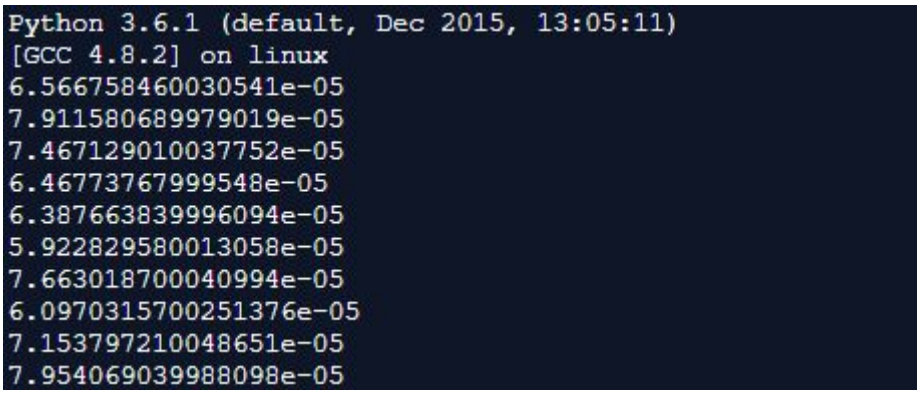
```python
import timeit

def linear_search(l, e):
  k = 0
  while k < len(l):
    if l[k] == e:
      return True
    else:
      k = k + 1
  return False

def linear_tempo_menor_numero():
  SETUP = '''
from __main__ import linear_search'''

  TESTE = '''
lista = []
for i in range(1, 101):
  lista.append(i)
linear_search(lista, 1)
  '''
  tempo = timeit.timeit(setup = SETUP, stmt = TESTE, number = 10000)
  print(tempo/10000)

def linear_tempo_maior_numero():
  SETUP = '''
from __main__ import linear_search'''

  TESTE = '''
lista = []
for i in range(1, 101):
  lista.append(i)
linear_search(lista, 100)
  '''
  tempo = timeit.timeit(setup = SETUP, stmt = TESTE, number = 10000)
  print(tempo/10000)

def linear_tempo_valores_aleatorios():
  SETUP = '''
from __main__ import linear_search
from random import randint'''

  TESTE = '''
lista = []
for i in range(1, 101):
  lista.append(i)
valor = randint(1, 100)
linear_search(lista, valor)
  '''
  tempo = timeit.timeit(setup = SETUP, stmt = TESTE, number = 10000)
  print(tempo/10000)

if __name__ == '__main__':
  linear_tempo_menor_numero()
  linear_tempo_maior_numero()
  linear_tempo_valores_aleatorios()
```

No código acima, as funções **linear_tempo_menor_numero()** , **linear_tempo_maior_numero()** e **linear_tempo_valores_aleatorios()** , respectivamente, contabilizam o tempo médio de execução da função **linear_search()**, considerando que João pensou no menor número, no maior número, e para 100 casos aleatórios; e executando cada teste 10,000 vezes. Os resultados do tempo médio de execução da função **linear_search()** para cada caso, em segundos, são apresentados abaixo:

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
2.334210629996696e-05
6.845083090011031e-05
5.032080750024761e-05
```

Nos resultados acima, podemos observar que para o caso de João ter pensado no número 1 (linha 1 dos resultados, representado o **melhor caso**), o tempo de execução, como esperado, é mais rápido do que se ele tivesse pensado 100 (linha 2 dos resultados, representado o **pior caso**). Para o caso de teste com números aleatórios (linha 3 dos resultados), o tempo foi maior do que para melhor caso, e menor do que o pior caso.

Até aí, tudo bem. O problema é que se executarmos esse código novamente, teremos resultados diferentes! Na imagem abaixo, podemos ver uma sequência de 10 execuções dos testes com números aleatórios.



A pergunta é: por que isso acontece? Há diversos fatores que influenciam o tempo de execução do código, como processos em background sendo executados na máquina, o algoritmo utilizado para gerar os números aleatórios, entre outros. Ao utilizarmos essa abordagem, para termos resultados robustos, é necessário ter conhecimento em design de estudos empíricos e análise estatística. E isso não é tudo! Há ainda mais fatores que precisamos levar em consideração: a linguagem de programação, compilador e processador usados e a representatividade dos casos de entrada definidos.

Por exemplo, como você consegue garantir que os casos de entradas que você usou sejam realmente representativos? Quem garante que *você* implementou *bem* os algoritmos? Um mesmo algoritmo pode ser implementado usando iterações com for, while ou até recursão (em cauda ou não)! Ou seja, é muito difícil realizar uma análise certeira, pois são muitos fatores para serem levados em consideração!

Outra desvantagem é que a gente só pode realizar a análise ***depois*** de ter implementado o algoritmo. =(

Será que é necessário a gente mensurar o tempo de execução de um código para avaliar a eficiência de um algoritmo? Será que não tem uma

abordagem mais simples e inteligente? A resposta é: SIM! Para isso, vamos modelar o problema como um de Contagem e utilizar o conceito de notação assintótica, explicado na próxima postagem.

Sumário

Nesta postagem, você aprendeu sobre:

- Um algoritmo simples para realizar a busca de um elemento: a **busca linear**.
- **Mensurar o tempo de execução** é uma opção, mas **não** é a melhor para analisar a **eficiência de um algoritmo**.
- A eficiência de um algoritmo depende das entradas. Nós vimos exemplos de **melhor caso** e **pior caso** para uma busca linear.