Análise de Algoritmos: Contagem de Instruções

Como foi apresentado na postagem anterior sobre Análise de Algoritmos (se não leu ainda, leia aqui), o problema de analisar a eficiência de algoritmos pode ser modelado como um problema de Contagem.

Nesse caso, o problema é contabilizar quantas instruções são executadas pelo algoritmo e, de acordo com os seus respectivos custos, calcular o custo total do algoritmo. O custo de uma instrução e o conceito em si de instrução depende da máquina. Por exemplo, em uma máquina a instrução de deslocamento de bit pode ter custo ***1***, e em outra, ***1,5***. Um incremento no valor de uma variável (exemplo, em ***Java*** seria ***i++***), em uma máquina poderia ser considerada uma instrução, e em outra, duas. Para simplificar, vamos assumir que cada linha representa uma instrução e o custo é o mesmo para qualquer instrução. Como veremos mais adiante, essas informações serão irrelevantes para a análise de algoritmos.

Considere o algoritmo de busca linear 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
```

Vamos assumir que ***l*** é uma lista contendo todos os números de inteiros de 1 até 100. Para ***e = 1***(**melhor caso**), temos que o custo é:

- linha 1 executada -> custo t
- linha 2 executada -> custo t
- linha 3 executada -> custo t
- linha 4 executada -> custo t
- linha 5 executada -> custo t
- Total = 5t

Para ***e = 100*** (**pior caso**), temos que o custo é:

- linha 1 é executada apenas uma vez -> custo t
- linha 2 é executada apenas uma vez -> custo t
- linhas 3, 4, 6 e 7 são executadas 99 vezes -> custo 99*(4t) = 396t
- linha 8 é executada 1 vez -> custo t
- Total = 399t

Com essa análise, como esperado, podemos concluir que com ***e = 100*** o custo de execução é bem maior quando comparado com ***e = 1***. Como analogia, podemos comparar esse processo de contagem com o

número de passos que você precisaria executar caso rodasse o código com um depurador (*debugger*).

Apesar desta abordagem ser mais simples do que mensurar o tempo de execução (como vimos na postagem passada), ela ainda tem um **problema grave**: o custo depende do formato da entrada. Ou seja, mesmo a gente apenas considerando o caso de termos uma lista ***l*** com 100 elementos, dependendo do valor de ***e***, a análise é realizada de forma diferente. Além disso, não podemos esquecer que fizemos várias simplificações para facilitar a análise! Então, o ponto é: será que é necessário a gente saber exatamente o custo de cada algoritmo? A resposta é: NÃO!

Na verdade, para entradas pequenas, o algoritmo pode até se tornar irrelevante. Por exemplo, para o mesmo problema, temos um algoritmo mais rápido (apenas acredite em mim, depois vamos analisá-lo): a **busca binária**.

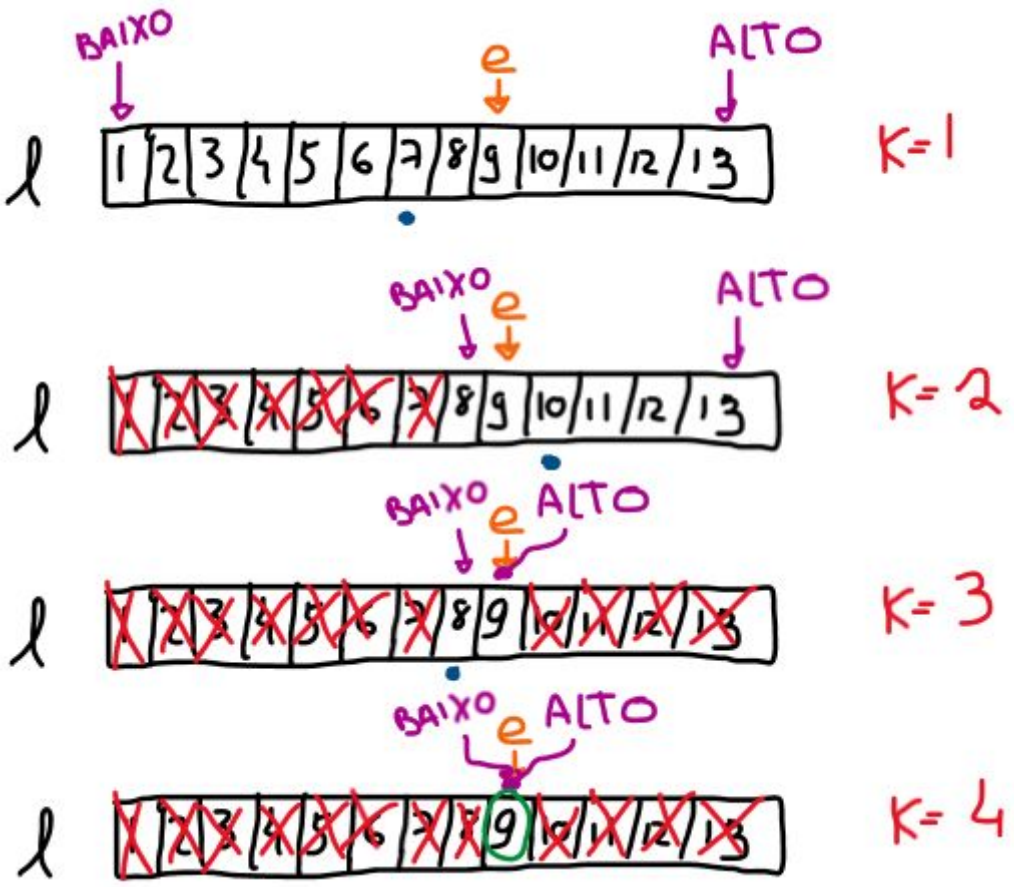
Para ilustrar o funcionamento deste algoritmo, vamos utilizar o mesmo exemplo da postagem passada: o jogo da adivinhação de números, utilizado na postagem anterior (se não conhece o jogo, veja aqui). Vamos supor que João pensou no número 75. Maria iniciaria chutando o número “do meio”, nesse caso, seria 50. João responderia “BAIXO”, indicando para Maria que o chute dela foi baixo. Com isso, Maria já sabe que o número não pode ser menor do que 50, assim, eliminando metade das opções! O próximo chute de Maria seria o número entre 51 e 100, no caso, 75. Com isso, com 2 chutes, ela acertaria! Se ela tivesse usado a busca linear, precisaria de 75 chutes!

Abaixo o pseudocódigo da busca binária é apresentado, seguido pela sua implementação em Python (que você pode acessar em um browser online aqui).

```
1 proc buscabinaria(lista l, elemento e)
2     baixo = 0
3     alto = tamanho de l - 1
4     enquanto baixo for menor ou igual alto faca
5         meio = (baixo + alto) / 2
6         chute = l[meio]
7         se chute == e
8             retornar VERDADEIRO
9         caso contrário, se chute > e
10            alto = meio - 1
11        caso contrário
12            baixo = meio + 1
13     retornar FALSO
```

O raciocínio do pseudocódigo acima é simples. Na busca binária, a gente sempre testa o valor central (linha 5) do intervalo que a gente está buscando. Dado que a gente vai sempre tentar dividir o problema pela metade, a gente precisa de uma variável para o limite inferior (linha 2) e uma para o limite superior (linha 3). Esses limites representam a “janela” que a gente usa para buscar na lista. Inicialmente, o intervalo é a lista inteira (linhas 2 e 3). A gente deve executar o laço (linha 4) até que a gente encontre o valor (linha 7) ou a nossa “janela de busca” quebre (***baixo > alto***). Caso o valor encontrado seja maior que o elemento em questão (linha 9), isso significa que o chute foi muito alto, e devemos reduzir o espaço de busca reduzindo o valor de ***alto*** (linha 10). Caso o valor encontrado seja menor que o elemento em questão (linha 11), isso significa que o chute foi muito baixo, e devemos reduzir o espaço de busca aumentando o valor de ***baixo*** (linha 12).

Na Figura abaixo, eu ilustro o caso de João ter pensado no número 9, em uma lista de 1 a 13.

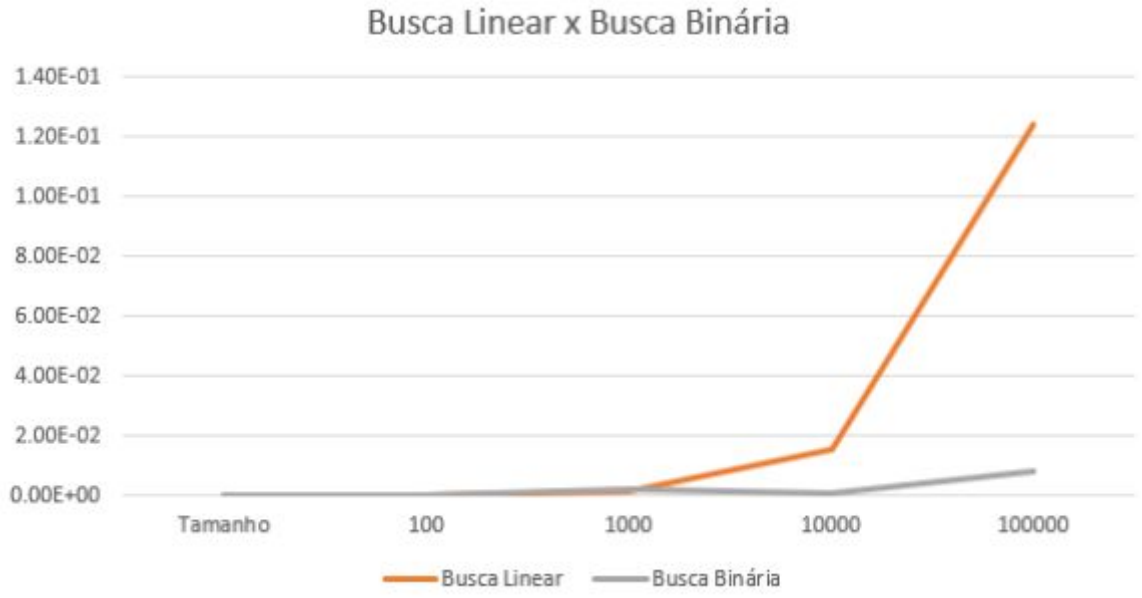


Na Figura acima, K representa o número de chutes até o momento, e a bolinha azul representa o último chute dado. Pode-se observar que são necessários 4 chutes (comparações) para acertar (encontrar) o número 9 em uma **busca binária** considerando uma lista de 1 a 13. Caso tivessemos utilizado uma **busca linear** iniciando de 1, precisaríamos de 9 chutes.

Abaixo, apresento o código em Python para a **busca binária**.

```
1  def binary_search(l, e):
2    baixo = 0
3    alto = len(l) - 1
4    while(baixo  e):
5        alto = meio - 1
6      else:
7        baixo = meio + 1
8    return False
```

Neste ponto, eu espero que vocês tenham, pelo menos, intuitivamente, a impressão de que, no geral, a **busca binária** é mais rápida do que a **busca linear**. De qualquer forma, podemos visualizar ao instrumentar o código para calcular o tempo de execução de cada algoritmo. Com isso, temos uma ideia de sua eficiência (como apresentado na postagem passada, essa não é uma forma tão confiável de análise!). Abaixo, eu apresento uma tabela com os tempos de execução, considerando a média de 10,000 execuções de cada algoritmo considerando que ***e*** é definido aleatoriamente para listas com diversos tamanhos:



Como podemos observar na figura acima, a eficiência do algoritmo só começa a ter diferença para listas com mais de 1000 elementos, e mesmo assim, na prática, podemos ver que ambos algoritmos são

quase equivalentes até 10000! Ou seja, isso significa que **a análise de algoritmos só interessa quando a gente está lidando com problemas de larga escala**! Ou seja, o que **realmente importa**é avaliar a **taxa de crescimento do custo de execução dos algoritmos** em função do tamanho do problema.

Então, será que não tem uma forma mais simples de analisar a taxa de crescimento ignorando os efeitos constantes (os custos de cada instrução) e levando em consideração apenas o tamanho do problema? A resposta é, felizmente, SIM! Para tal, pegamos emprestado dos matemáticos o conceito de **notação assintótica**, tópico da próxima postagem.

Sumário

Nesta postagem, você aprendeu sobre:

- Um algoritmo eficiente para realizar a busca de um elemento: a **busca binária**.
- A **análise de algoritmos** só se torna relevante quando estamos lidando com **problemas de larga escala**.
- Avaliar a eficiência de um algoritmo **contabilizando o custo de execução das instruções** é melhor do que mensurar o tempo de execução.
- Na análise de algoritmos, o que importa é a **taxa de crescimento do custo de execução dos algoritmos** em função do tamanho do problema.