Análise de Algoritmos com Laços e Condicionais

Em postagens passadas, eu apresentei os conceitos básicos para a análise de algoritmos, a análise de algoritmos como contagem de instruções, a notação assintótica, e a prova de que a análise assintótica faz o que se propõe (ignorar termos de ordem inferior e fatores constantes.

Nesta postagem, eu vou apresentar, por meio de exemplos, como realizar a análise de algoritmos com laços e condicionais utilizando a **notação O**. Serão discutidas as classes de complexidade O(1), O(n), O(n^c), O(logn) e O(loglogn). Posteriormente, eu apresento como combinar laços consecutivos e lidar com condicionais.

Classes de complexidade

O(1)

Esse classe é denominada **Constante** e é a de menor complexidade assintótica, e o desejo inalcançavel para a maioria dos algoritmos. Um trecho de código se encaixa nesta classe caso não contenha laços com quantidade de execução variável, recursões e não chame outra função com tempo não-constante.

Por exemplo, considere o procedimento para trocar os valores armazenados nas variáveis **x** e **y**apresentado abaixo:

```
1  proc trocaValores(x, y)
2     temp = x
3     x = y
4     y = temp
```

No código acima, os valores das variáveis x e y são trocados. As linhas 2, 3, e 4 apenas contém instruções para alocar valores em variáveis, que tem custo constante, ou seja, **O(1)**. Na análise de algoritmos, para medir o custo de expressões sequenciais, aplica-se a **Regra da Simplificação da Soma**. Segundo a Regra da Simplificação da Soma, dado **f1(n) = O(g1(n))** e **f2(n) = O(g2(n))**, então:

$$f_1(n) + f_2(n) = O\left(\max\left(g_1(n), g_2(n)\right)\right)$$

Dado isso, temos que o custo total do procedimento acima é **O(max(O(1),O(1),O(1))**; ou seja, **O(1)**.

Outro caso seria a um laço que é executado um número constante de vezes. Um exemplo é apresentado abaixo.

```
1  # c é um valor constante
2  inicializar k = 1; repetir enquanto k < c; incrementando k = k + 1
3     # algumas expressões O(1)
```

Neste caso, a linha 2 contém um laço com valor constante, tendo complexidade O(1). As expressão representadas na linha 3 estão aninhadas com o laço, tendo complexidade O(1). Na análise de algoritmos, para medir o custo de expressões aninhadas, aplica-se a **Regra da Simplificação do Produto**. Segundo a Regra, dado **f1(n) = O(g1(n))** e **f2(n) = O(g2(n))**, então

$$f_1(n) * f_2(n) = O\left(g_1(n) * g_2(n)\right)$$

Desta forma, temos que o custo total é **O(O(1)*O(1))**, resultando em **O(1)**.

O(n)

A complexidade de tempo de um laço é considerado **O(n)**, ou seja, **Linear**, se a variável de controle do mesmo é incrementada ou decrementada por um valor constante. Exemplos abaixo:

```
1  # c é uma constante
2  inicializar k = 1; repetir enquanto k < n; incrementando k = k + c
3     # algumas expressões O(1)
4  inicializar k = 1; repetir enquanto k < n; incrementando k = k - c
5     # algumas expressões O(1)
```

No exemplo acima, ambos os laços (linhas 2 e 4) são **O(n)**.

O(n^c)

No contexto de laços, a classe de complexidade **polinomial** ocorre como consequência da aplicação da **Regra da Simplificação do Produto** (discutida acima, na classe **O(1)**), quando temos múltiplos laços com complexidade **O(n)** aninhados. Por exemplo, abaixo são apresentados casos nos quais a complexidade é **O(n^2)**.

```
1  # c é uma constante
2  inicializar k = 1; repetir enquanto k < n; incrementando k = k + c
3      inicializar j = 1; repetir enquanto j < n; incrementando j = j + c
4         # algumas expressões O(1)
5  inicializar k = 1; repetir enquanto k < n; incrementando k = k - c
6      inicializar j = n; repetir enquanto j > 0; incrementando j = j - c
7          # algumas expressões O(1)
```

No exemplo acima, ambos os laços aninhados (linhas 2 e 3; e linhas 5 e 6) são **O(n^2)**.

O(logn)

A complexidade de tempo de um laço é considerado **O(logn)**, ou seja, **Logarítmica**, se a variável de controle do mesmo é multiplicada ou dividida por um valor constante. Exemplos abaixo:

```
1  # c é uma constante
2  inicializar k = 1; repetir enquanto k < n; incrementando k = k * c
3     # algumas expressões O(1)
4  inicializar k = 1; repetir enquanto k < n; incrementando k = k / c
5     # algumas expressões O(1)
```

No exemplo acima, ambos os laços (linhas 2 e 4) são **O(logn)**. Mais abaixo, eu explico em detalhes o porquê da complexidade ser logarítmica, ao revisar o conceito de logaritmo!

O(loglogn)

A complexidade de tempo de um laço é considerado **O(loglogn)**, ou seja, **Loglogarítmica**, se a variável de controle do mesmo é reduzida ou aumentada exponencialmente por um valor constante. Exemplos abaixo:

```
1  # c é uma constante maior do que 1
2  inicializar k = 1; repetir enquanto k < n; incrementando k^c
3     # algumas expressões O(1)
4  # b é uma constante entre maior do que 0 e menor do que 1
5  inicializar k = 1; repetir enquanto k < n; incrementando k^b
6     # algumas expressões O(1)
```

No exemplo acima, ambos os laços (linhas 2 e 5) são **O(loglogn)**.

Combinando laços consecutivos

Quando há laços consecutivos, nós aplicamos a **Regra da Simplificação da Soma**, apresentada acima ao discutir a classe O(1). Dado isso, vamos analisar alguns exemplos.

```
1  inicializar k = 1; repetir enquanto k < n; incrementando k = k + 1
2     # algumas expressões O(1)
3  inicializar k = 1; repetir enquanto k < n; incrementando k = k + 1
4     # algumas expressões O(1)
```

No pseudocódigo acima, a complexidade das linhas 1 e 3 são, respectivamente, **O(n)** e **O(n)**. Então, temos que a **complexidade total** é **max(O(n),O(n))**, resultando em **O(n)**.

Agora, vamos analisar o exemplo abaixo:

```
1  inicializar k = 1; repetir enquanto k < n; incrementando k = k + 1
2     # alguma expressões O(1)
3  inicializar k = 1; repetir enquanto k < n; incrementando k = k + 1
4      inicializar j = 1; repetir enquanto j < n; incrementando j = j + 1
5         # algumas expressões O(1)
```

No pseudocódigo acima, a complexidade das linha 1 é **O(n)**. Nas linhas 3 e 4 temos dois laços aninhados, e nesse caso, a complexidade é **O(n^2)**. Então, temos que a **complexidade total** é **max(O(n),O(n^2))**, resultando em **O(n)^2**.

Lidando com condicionais

Quando analisamos um laço com condicionais, é necessário levar com consideração que precisamos considerar o **pior caso**. Dessa forma, precisamos avaliar quando as condicionais influenciam no número de iterações a serem executadas. Para exemplificar, vamos utilizar dois algoritmos clássicos: **busca linear** e **busca binária**.

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
```

O objetivo é definir o custo total do procedimento declarado na linha 1. A linha 2 é O(1). Analisando o código, o pior caso seria nunca executar a linha 5, e, consequentemente, executar a linha 8. Ou seja, as condicionais não reduzem o número de iterações a serem executadas. Como consequência, a complexidade de tempo da **busca linear** é **O(n)**.

No caso da busca binária, o comportamento é diferente. O pseudocódigo da busca binária é apresentado abaixo:

```
proc buscabinaria(lista l, elemento e)
    baixo = 0
    alto = tamanho de l - 1
    enquanto baixo for menor ou igual alto faca
    meio = (baixo + alto) / 2
    chute = l[meio]
    se chute == e
        retornar VERDADEIRO
    caso contrário, se chute > e
        alto = meio - 1
    caso contrário
        baixo = meio + 1
     retornar FALSO
```

Ao analisar o algoritmo, percebe-se que, mesmo que o elemento não esteja na lista (o pior caso), a cada iteração, o **problema é reduzido pela metade**. Isso é operacionalizado pelo uso das variáveis **baixo** e **alto** para controlar o espaço de busca na lista. Então, qual é a função matemática que representa essa redução do número de iterações?

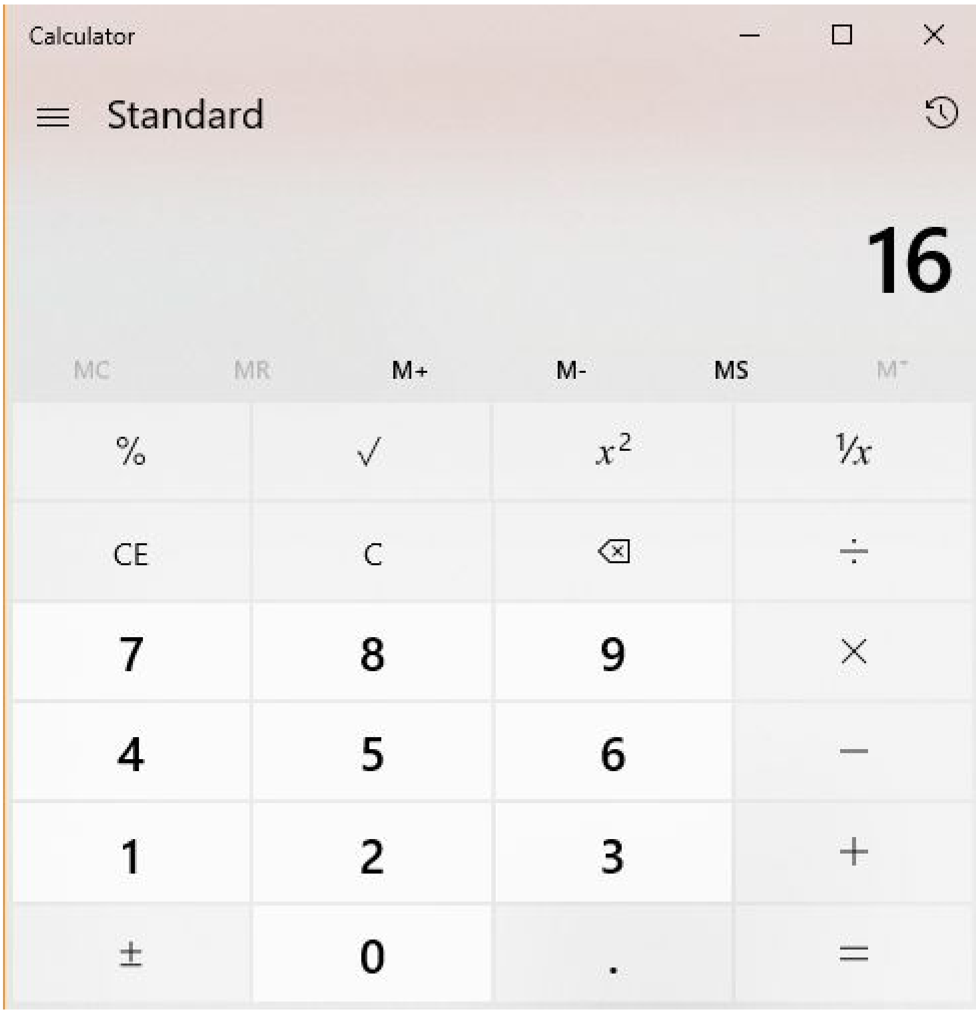
A resposta é: logaritmos! Apesar de muitas pessoas terem dificuldades e até se assustarem ao trabalhar com logaritmos, a sua definição prática é muito simples. Antes de explicar, vamos ver um exemplo:

$$\log_2 16$$

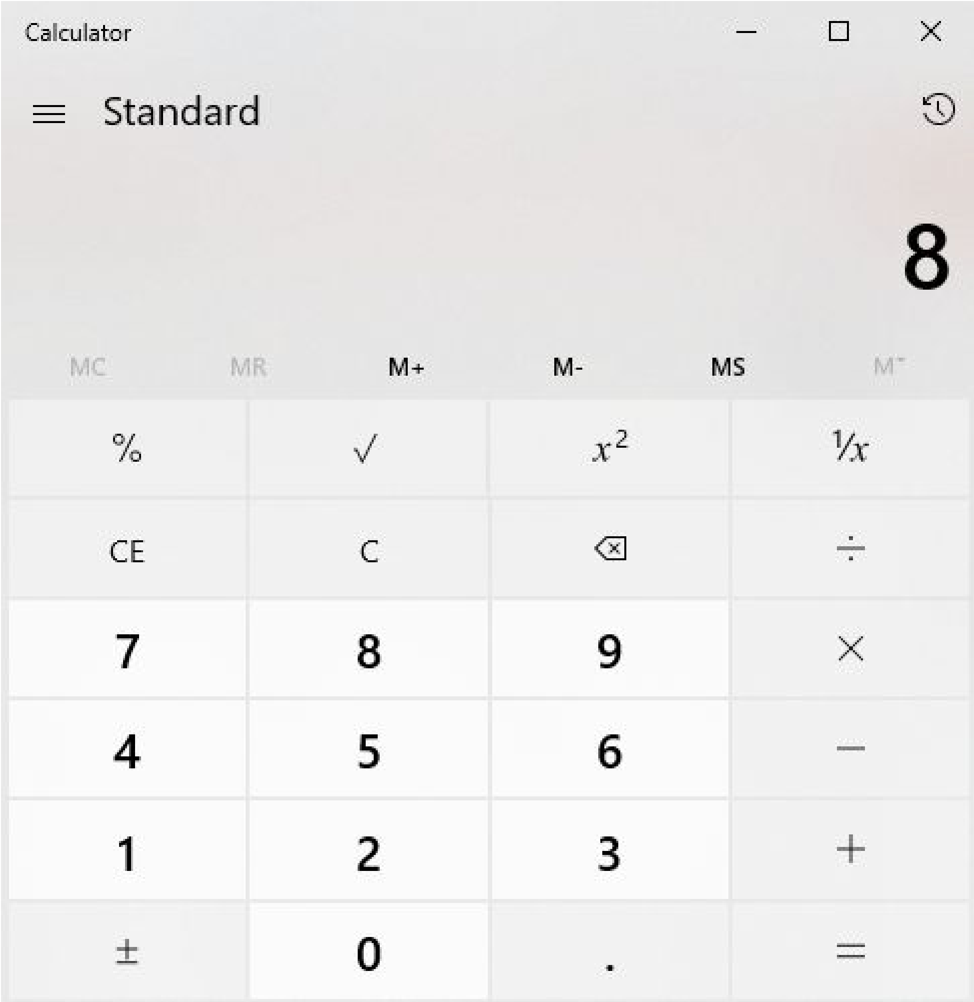
Leia-se: logaritmo de 16 na base 2.

Se você executar esse cálculo em uma calculadora científica, poderá verificar que o resultado é 4.

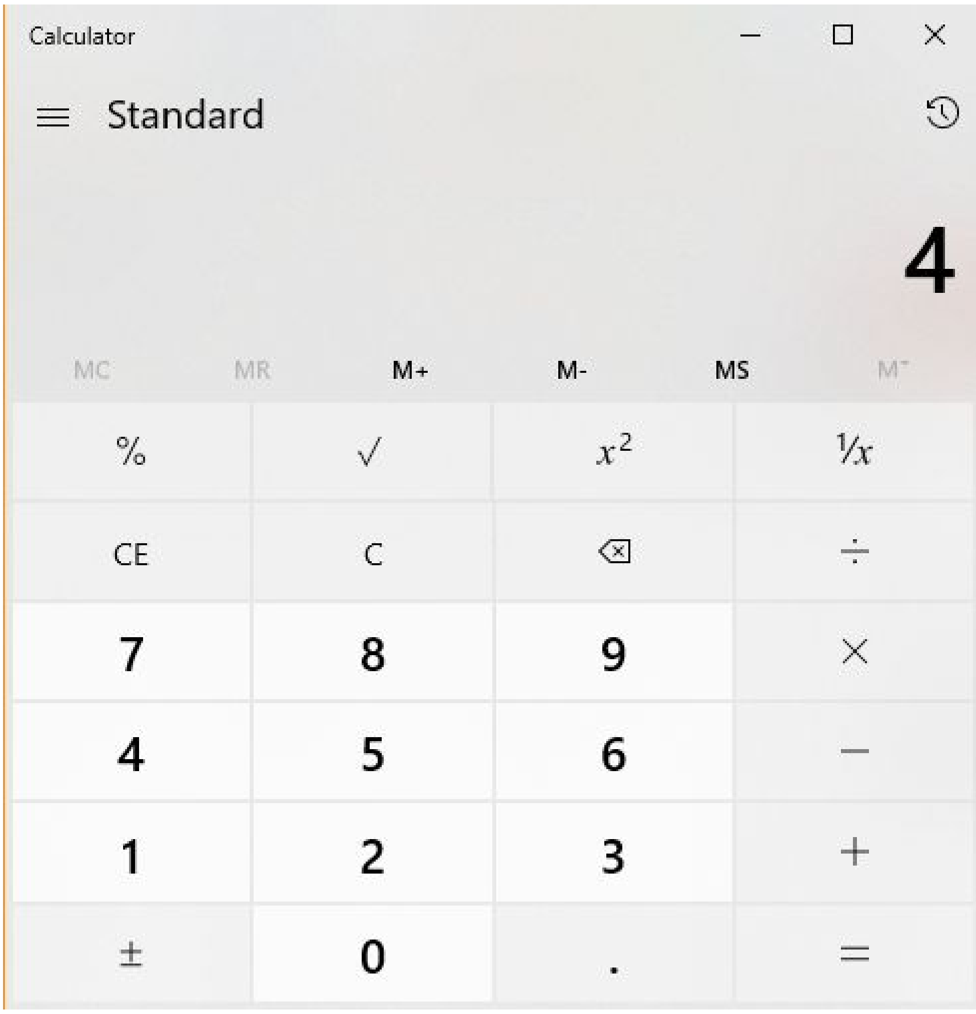
Agora, pegue uma calculadora comum. Escreva 16:



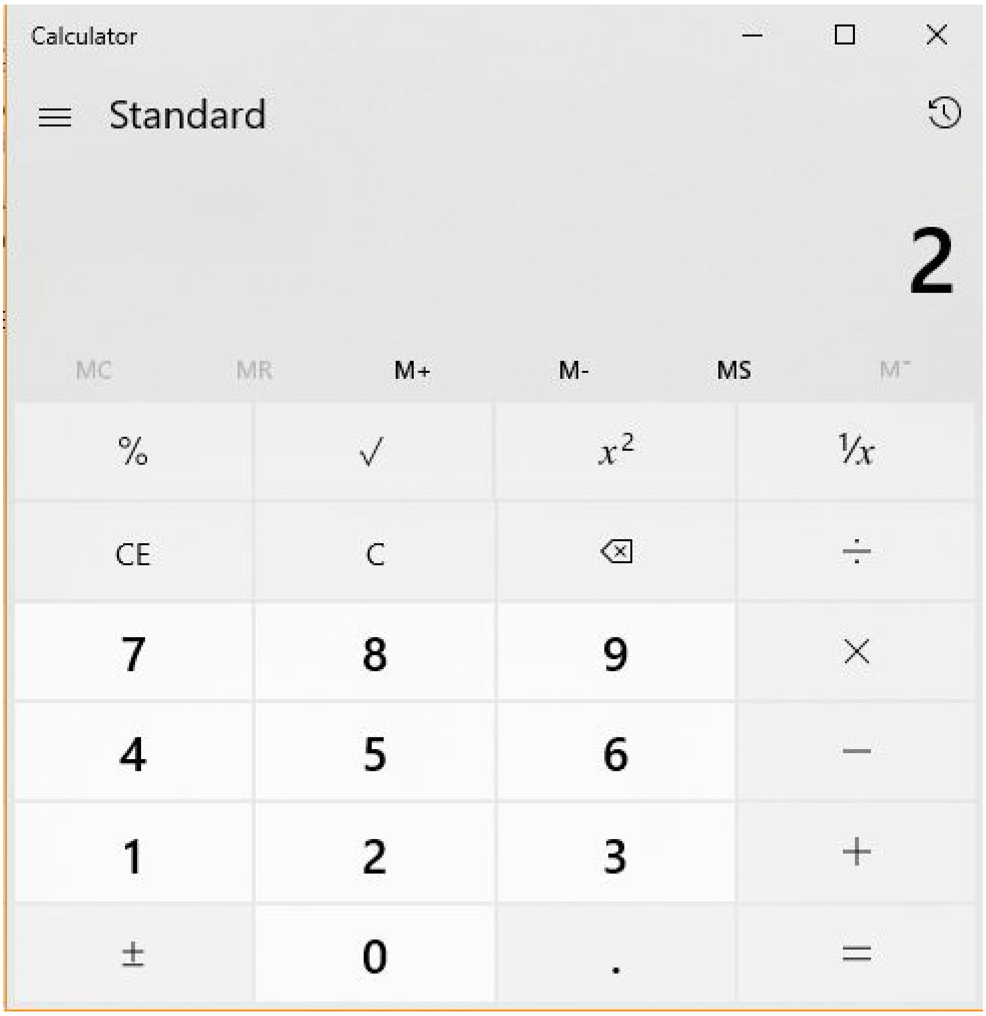
Agora, divida por 2:



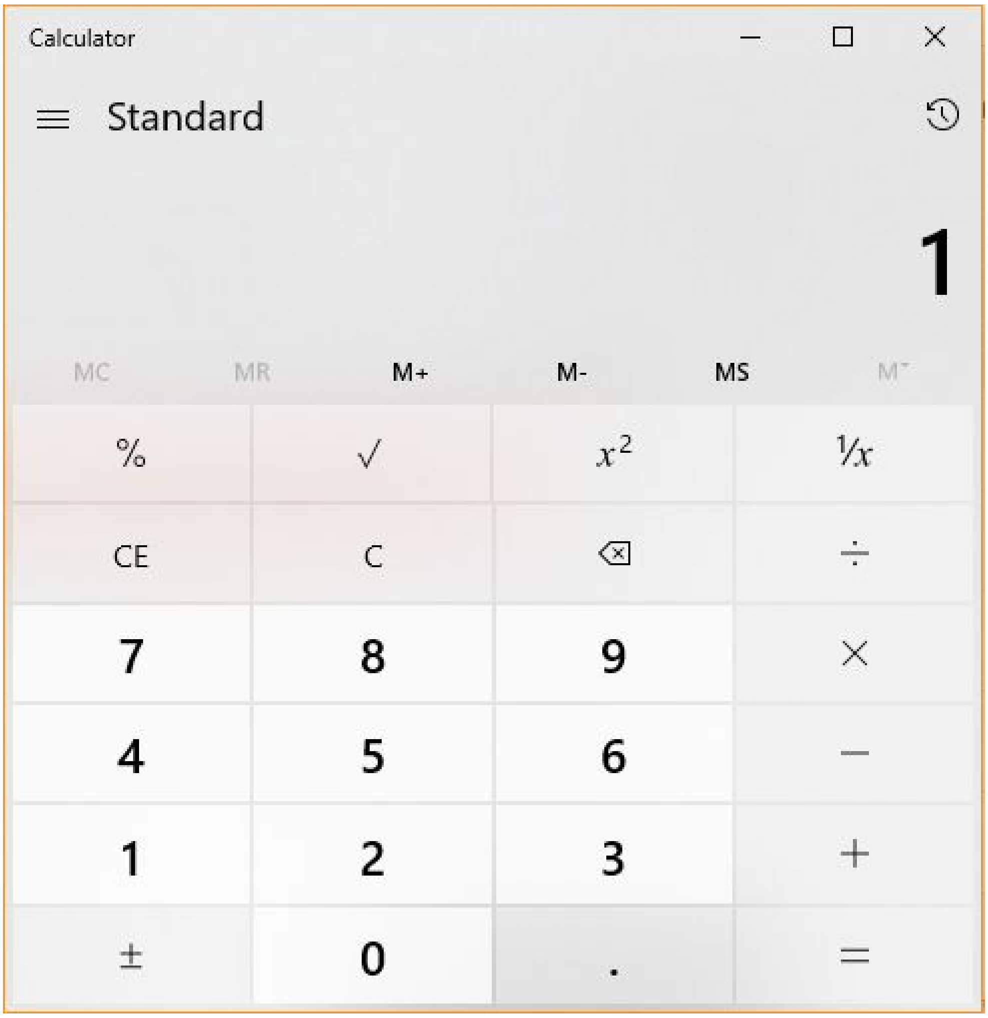
Novamente, divida por 2:



Novamente, divida por 2:



Novamente, divida por 2:



Pronto! Pode parar agora. Chegamos em 1. **Quantas vezes você apertou o botão de divisão?** Não coincidentemente, **foram 4 vezes**! Por que? O **logaritmo de n na base b**, na prática, significa: **quantas vezes eu posso reduzir um número n, de acordo com uma base b, até chegar em 1** (ou um número menor do que 1). Ou seja, no nosso contexto, **“quantas vezes eu posso dividir o problema?”**. No caso da busca binária, sempre reduzimos pela metade. Ou seja, estamos sempre reduzindo a lista por um fator 2 (base 2)! Por isso, a complexidade da busca binária é, **O(log(2)n)**(leia-se, log de n na base 2). A base é irrelevante na análise assintótica, e para simplificar, podemos dizer que é **O(logn)**.

Geralmente, as condicionais ou não influenciam no influenciam no número de iterações para o pior caso (é o caso da busca linear), ou reduzem esse número. Então, no caso das expressões condicionais serem muito complexas de serem avaliadas, uma opção é ignorá-las e definir um limite superior assintótico apenas avaliando a expressão do laço em si. Com esta abordagem, corre-se o risco do limite superior definido não ser o mais próximo o possível, mas pode ser bom o suficiente para a análise.

Na próxima postagem, vamos falar sobre análise de algoritmos recursivos.

Sumário

Nesta postagem, você aprendeu sobre:

- Analisar algoritmos com laços de diversas classes de complexidade de tempo: **O(1)**, **O(n)**, **O(n^c)**, **O(logn)**, e **O(loglogn)**.
- Analisar algoritmos com laços e condicionais.
- As complexidades de tempo da **busca linear** e da **busca binária** são, respectivamente, **O(n)** e **O(logn)**.
- Uma definição simples para o entendimento da importância do logaritmo na análise de algoritmos, significando **“quantas vezes eu posso dividir o problema?”** Isso é **MUITO**importante!