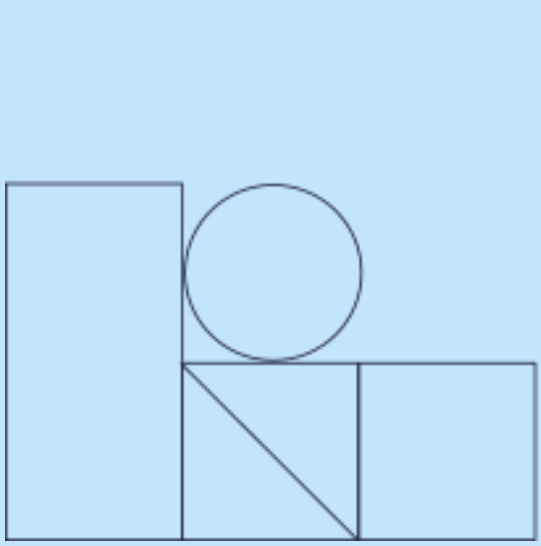
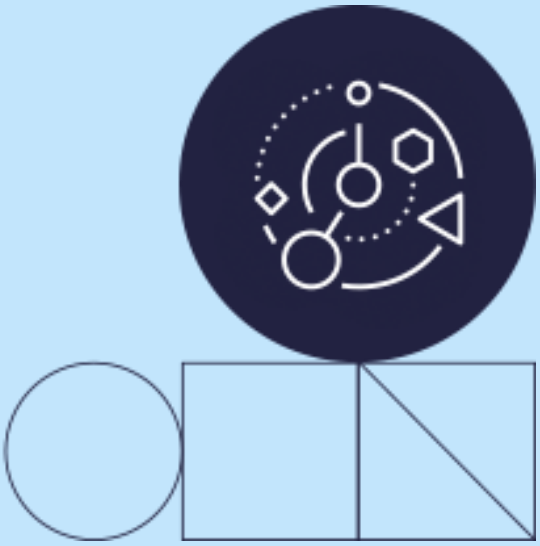
IBM SkillsBuild

IBM

Programação em Python

Complexidade dos algoritmos





Índice

Introdução

Quando falamos sobre a complexidade de um algoritmo, estamos na verdade nos referindo a uma métrica teórica que mede a eficiência computacional de um algoritmo. Se temos vários algoritmos que resolvem o mesmo problema, então esta métrica nos ajudará a definir qual dos algoritmos é computacionalmente melhor.

Embora não seja um conceito difícil de entender, a obtenção e o estudo da complexidade requer certas habilidades matemáticas. Nesta seção vamos expor de forma simples a complexidade algorítmica, os tipos de complexidade e os exemplos em código Python.

O que é um algoritmo?

Antes de mais nada, vamos falar de Algoritmo. Um algoritmo é uma seqüência de instruções cujo objetivo é resolver um problema. Há muitos problemas que têm algoritmos para resolvê-los. Além disso, o mesmo problema pode ter várias soluções (ou nenhuma?). Se tivermos várias soluções para o mesmo problema, qual seria a melhor solução? Para isso, precisamos avaliar esse algoritmo, o resultado da avaliação o chamaremos de complexidade algorítmica.

Para medir um algoritmo, podemos tratá-lo de dois pontos de vista. Ou aquele que resolve o problema em menos tempo ou aquele que utiliza menos recursos do sistema. Chamaremos a idéia de complexidade de tempo e a idéia de recursos do sistema que chamaremos de complexidade espacial.

Dos dois tipos mencionados, o menos relevante é a complexidade espacial, devido aos altos recursos que uma máquina pode ter. Além disso, o tempo é muito mais valioso. Podemos dizer que, dos dois, o tempo é o único que não pode ser comprado. Portanto, quando nos referimos à complexidade algorítmica, estamos nos referindo à complexidade temporal.

Basicamente, a medição do algoritmo é para medir quanto tempo leva para resolver um problema.

O tamanho importa?

Vamos falar de alguns conceitos que todos os algoritmos têm.

Durante seu processo, digamos um algoritmo de classificação, ele tem uma série de instruções que são repetidas, conhecidas como loops. Também uma série de escolhas ou comparações, (if.. else..) que a levam a seguir certas instruções ou a não seguir outras. Todos estes elementos fazem parte do algoritmo. Eles também têm dados de entrada e saída.

De todas essas características, as que podemos variar para medir nossa complexidade seriam os dados de entrada. Não é o mesmo, em um algoritmo de classificação, classificar 10 elementos do que classificar 1.000.000 de elementos. O algoritmo deve ser capaz de classificar os elementos independentemente do tamanho do vetor, mas tem um impacto direto no tempo que leva para resolver o problema.

Com isto podemos começar a medir os diferentes algoritmos de classificação. Se classificarmos um vetor de dez milhões com vários algoritmos e vermos qual deles leva menos tempo, podemos dizer qual deles tem a maior ou menor complexidade algorítmica.

A complexidade algorítmica não é um número. É uma função

Agora, vamos considerar que experimentamos o algoritmo com 10.000.000 de entradas. Se o executarmos em um computador muito potente, a resposta do programa seria obviamente muito mais rápida do que em um computador com poucos recursos. Portanto, o algoritmo sempre terá um tempo de resposta diferente para cada computador.

Para resolver isto, não mediremos o tempo necessário para que um algoritmo responda. Ao contrário, contaremos o número de instruções, assumindo que cada instrução seja executada no mesmo tempo constante. Desta forma, mediremos quantas instruções o algoritmo leva para resolver o problema no que diz respeito ao tamanho do problema.

Vejamos o seguinte código escrito em Python:

```
def codigo_1( number ):
  a = 0
  for j in range(1, number+1):
    a += a + j

  for k in range(number, 0, -1):
    a -= 1
    a *= 2
  return a
```

Neste algoritmo, temos algumas instruções e vários loops. Vamos começar com a contagem das instruções.

Temos uma atribuição da variável **a**, então temos 1 instrução.

Instruções = 1

O seguinte é um laço com uma instrução dentro. Dependendo do valor da variável **$number**, é realizada uma instrução **n** vezes. Por exemplo, se **$number** tem o valor de **3** então as instruções executadas são 3 vezes. Então nós temos:

Instruções = (1)+(n)

Instruções = n+1

Na segunda etapa acontece a mesma coisa. Mas desta vez há 2 instruções no loop do código. O número de instruções seria:

Instruções = 2n+n+1

Instruções = 3n+1

A complexidade algorítmica seria então **3n+1** porque é o número de instruções que tem que executar para resolver o problema.

O que é interessante é saber como a solução de um problema pode crescer em unidades de tempo. Neste exemplo, fica claro que o tempo cresce linearmente em relação ao valor de entrada.

Código e Complexidade

É possível calcular visualmente a complexidade de alguns algoritmos simples. Aqui estão alguns exemplos:

```
def codigo_2():
  a = 0
  a -= 1
  a *= 2
```

No Código 2, a Complexidade seria:

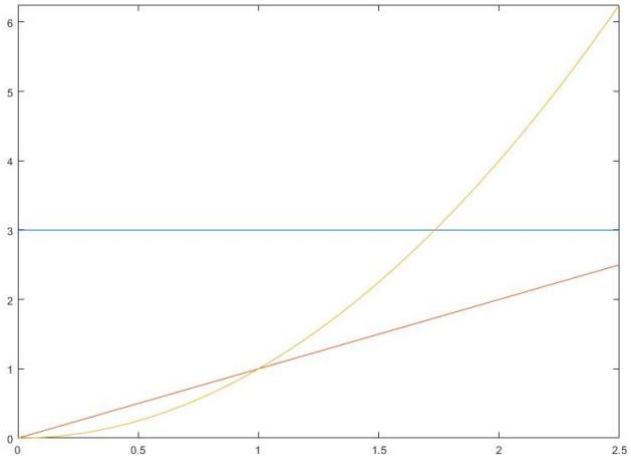
F(x) = 3

```
def codigo_3( number ):
  a = 0;

  for j in range(1, number+1):
    for k in range(1, number+1):
      a += a + ( k*j )

  return a
```

No código 3, temos um loop aninhado. Cada vez que você executa um laço, o outro é executado n vezes também. O que seria n vezes n. A complexidade seria então:

F(x) = n^2+1

Se traçarmos estas 3 funções, códigos 1, 2 e 3, podemos ver exatamente qual delas seria a de menor complexidade algorítmica:



Isto nos deixa com o fato de que, independentemente dos valores, é evidente que quanto mais dados, maior a curva se torna. Isto é o que estamos realmente interessados em ver na complexidade algorítmica, como o algoritmo se comporta com grandes volumes de insumos ao longo do tempo.

Pior cenário de caso

Alguns algoritmos podem ter tempos de execução diferentes, enquanto têm o mesmo tamanho de dados e os mesmos recursos computacionais. Isto porque, dependendo dos dados, às vezes uma solução pode ser alcançada na primeira iteração, ou todos os dados podem ter que ser atravessados.

O seguinte código visa encontrar o primeiro número par em uma lista de números:

```
def codigo_4(array):
  for k in range(len(array)):
    if( array[k] % 2 == 0 ):
      return k

  return null
```

Dependendo dos valores que são armazenados no vetor, assim será o tempo necessário para que o algoritmo resolva o problema. Se a matriz for uma seqüência de números inteiros, então será necessária apenas uma iteração. Se for uma lista de números ímpares, então passará por todas as iterações.

Na melhor das hipóteses, o número par será o primeiro da lista, o que concluiria o algoritmo. Na pior das hipóteses, não terá sequer um número par, pois passará por todas as instruções.

Para o melhor caso, teríamos uma complexidade algorítmica de:

F(x)=1

Para o pior caso, a complexidade algorítmica seria:

F(x)=n

Para expressar o pior caso, usaremos uma notação conhecida como “**O Grande**” e está escrito:

O(n)

Que significa **complexidade do pior caso**. Está escrito como “**O**” mas na verdade é a carta grega Omicron.

Ordem de Complexidade

Até este ponto, é possível medir qualquer algoritmo. Mas mesmo assim, é difícil comparar algoritmos uns com os outros. Além disso, é necessário ser capaz de categorizar quão complexo é um algoritmo em relação a outros. Para isso, recorreremos à ordem da complexidade.

Como vimos antes. Para medir um algoritmo, precisamos recorrer ao pior cenário possível e com um grande volume de dados.

Olhando assim, poderíamos simplificar as equações do Big-O, eliminando algumas características que, na entrada massiva de dados, são irrelevantes.

Se compararmos vários algoritmos, tais como:

O(3n+1)

O(20n)

O(15n+150)

O(n)

Podemos ver em seus gráficos que, independentemente da variável que corta o eixo Y ou a inclinação da curva, todos eles crescem com a mesma inclinação. Todas estas equações podem ser agrupadas e referidas como **ordem Linear**.

O mesmo pode ser dito das funções quadráticas, vejamos o seguinte grupo:

O(n^2+27)

O($20n^2$+3)

O(n^2+n+45)

O($100n^2$+101n)

Embora seus gráficos correspondentes tenham começos e posições diferentes, todos eles são uma parábola e terão um comportamento semelhante em relação ao aumento do número de entradas. Estes apresentam um comportamento muito diferente do conjunto anterior. Vamos chamar este grupo de grandes ORs de **ordem cuadrática**. Também podemos expressá-los como:

O($100m^2$+101n+) € O(n^2)

Desta forma, podemos classificar um algoritmo em um grupo particular e será muito mais fácil e rápido ver a complexidade de um algoritmo.

Quando somos advertidos da complexidade algorítmica de um algoritmo, podemos ter uma idéia de como ele irá se comportar.

Se, por exemplo, tivermos um algoritmo de ordem $O(100n^4 + 3n^3 + 53)$, então podemos definir sua ordem de qualquer uma das seguintes maneiras:

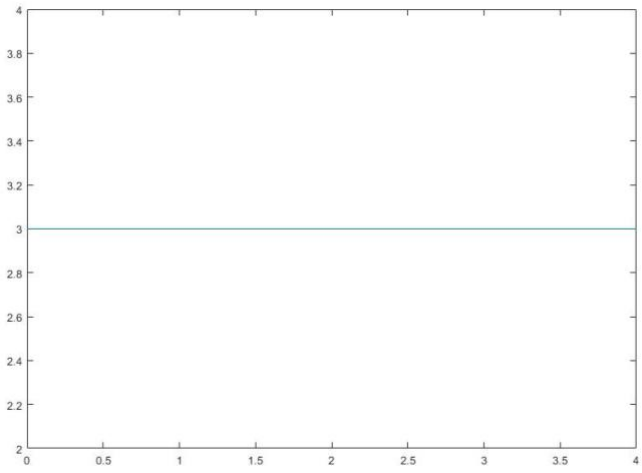
$$O(100n^4 + 3n^3 + 53) \subset O(100n^4 + 3n^3) \subset O(100n^4) \subset O(n^4)$$

Ordem de Complexidade Mais Conhecida

Constante: $O(1)$

É a mais simples e sempre tem um tempo de execução constante. Exemplo:

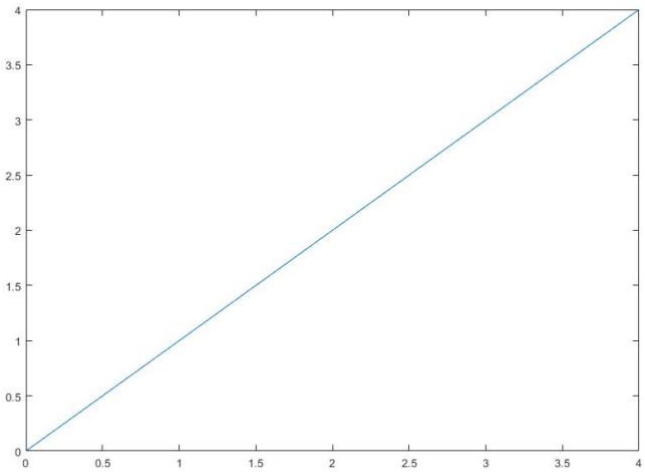
```
def constante():
  x = 50
  ++x
  return x
```



Linear: $O(x)$

O tempo cresce linearmente à medida que os dados crescem. Exemplo:

```
def lineal(number):
  result = 0
  for x in range(0, number):
    ++result

  return result
```

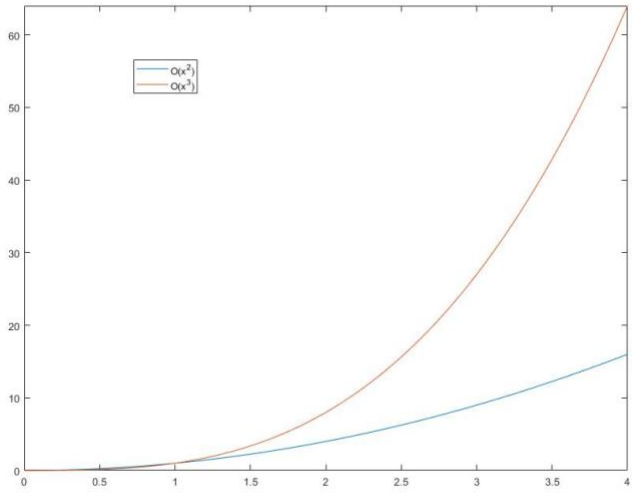


Polinômio: $O(x^c), c > 0$

Estes são os algoritmos mais comuns. Quando **c** é 2 é chamado quadrático, quando é 3 é chamado cúbico, e em geral, polinomial. Quando **n** é muito grande são muitas vezes muito complicadas. Estes algoritmos geralmente têm laços aninhados. Se eles tivessem 2 laços aninhados, seria um quadrático.

Exemplos:

```
def polinomico(number):
  x = 0
  for i in xrange(1,number):
    for j in xrange(1,number):
      x += i + j

  for i in xrange(1,number):
    for j in xrange(1,number):
      for k in xrange(1,number):
        x += i * j * k

  return x
```

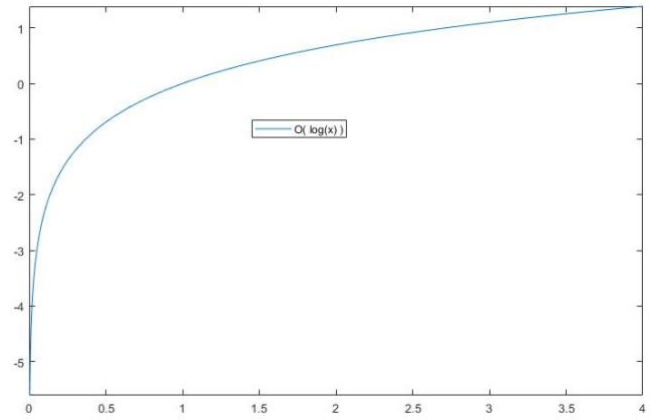


Logarítmico: $O(\log x)$

Normalmente não há muitos deles. Estes algoritmos indicam que o tempo é menor do que o tamanho dos dados de entrada. Não é importante indicar a base do logaritmo. Um exemplo é uma busca dicotômica.

Este algoritmo busca um elemento em uma matriz ordenada dividindo a matriz em 2 metades, identifica em qual das metades ele está, depois divide essa parte em 2 metades iguais e busca novamente até que o elemento seja encontrado, é um algoritmo recursivo:

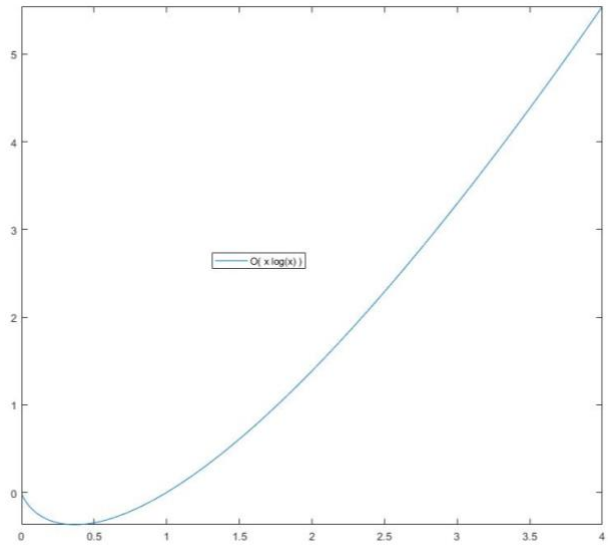
```
def bin(a,x,low,high):
  ans = -1
  if low==high: ans = -1
  else:
    mid = (low+((high-low)//2))
    if x < a[mid]: ans = bin(a,x,low,mid)
    elif x > a[mid]: ans =
bin(a,x,mid+1,high)
    else: ans = mid
  return ans
```



Enelogarítmico: $O(n \log x)$

Tão bom quanto o anterior, nesta ordem encontramos o algoritmo **QuickSort**. Um exemplo pode ser encontrado na Wikipedia, neste link https://en.wikipedia.org/wiki/Quicksort.

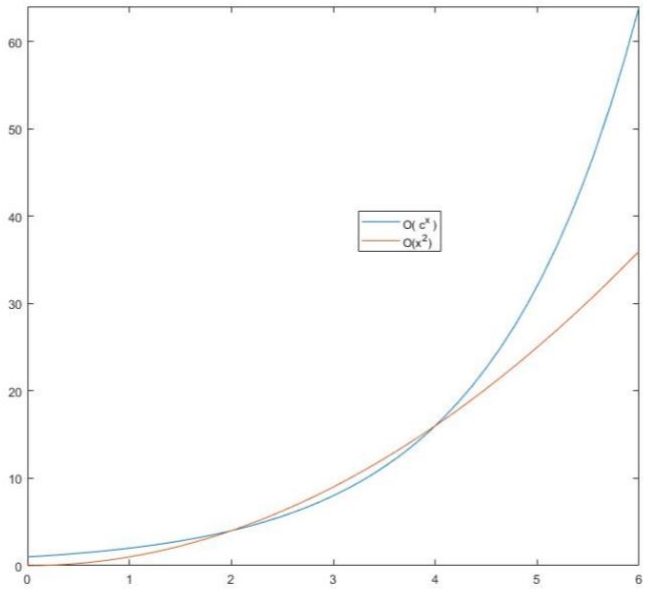
O código não está incluído devido às diferentes versões, estudos e discussões deste algoritmo. Mas vamos compartilhar o comportamento desta ordem de complexidade:



Exponencial: $O(c^x)$

É uma das piores complexidades algorítmicas. Ele sobe demais à medida que os dados de entrada crescem. Você pode ver na figura como tal função cresce. Um exemplo deste código é a solução Fibonacci, que gera 2 loops recursivos em cada execução. Exemplo:

```
def exponencial(n):
  if n==1 or n==2:
    return 1
  return exponencial(n-1)+exponencial(n-
2)
```



Comparação

Para destacar o nível de importância de categorizar um algoritmo em uma ordem de complexidade, temos que olhar para a extensão real em que os algoritmos podem ser complexos. Para compará-los uns com os outros, vamos supor que todos eles requerem 1 hora de computador para resolver um problema de tamanho N=100.

O(f(n))	N=100	t=2h	N=200
log n	1 h	10000	1.15 h
n	1 h	200	2 h
n log n	1 h	199	2.30 h
n^2	1 h	141	4 h
n^3	1 h	126	8 h
2^n	1 h	101	10^{30} h

Como podemos ver, à medida que a complexidade aumenta, o tempo necessário para completar a tarefa cresce muito, e pode aumentar enormemente em alguns casos, pois há mais dados para tratar.

Se encontrarmos uma função com complexidade quadrática (O(n2)) ou exponencial (O(2n)), geralmente será um sinal de que o algoritmo precisa de uma revisão urgente, e é melhor não utilizá-lo.

O interessante desta notação é que ela nos permite comparar vários algoritmos equivalentes sem nos preocuparmos com testes de desempenho que dependem do hardware utilizado. Isto é, quando confrontados com resultados equivalentes, podemos escolher o algoritmo de melhor desempenho, que será sempre o melhor independentemente do hardware se o conjunto de dados for suficientemente grande (em conjuntos de dados pequenos, hardware mais rápido pode dar resultados mais rápidos para um algoritmo menos eficiente, mas à medida que o conjunto de dados crescer, este não será mais o caso).

Assim, de agora em diante, quando virmos uma função documentando explicitamente sua complexidade usando a notação Big-O, prestaremos muita atenção e consideraremos as implicações de sua utilização em nossa aplicação. Da mesma forma, quando criamos um algoritmo para resolver um problema em nossa aplicação, é interessante observar na documentação (mesmo que esteja nos comentários do cabeçalho) qual é a complexidade algorítmica do algoritmo na notação Big-O. Isto ajudará qualquer programador que venha atrás dele, tanto para usá-lo como para ter que otimizá-lo, e ele saberá qual é a lógica a ser vencida.

Aqui está uma tabela muito completa com a complexidade dos principais algoritmos para estruturas de dados, ordenação de array, operações gráficas e operações de pilha (heap).