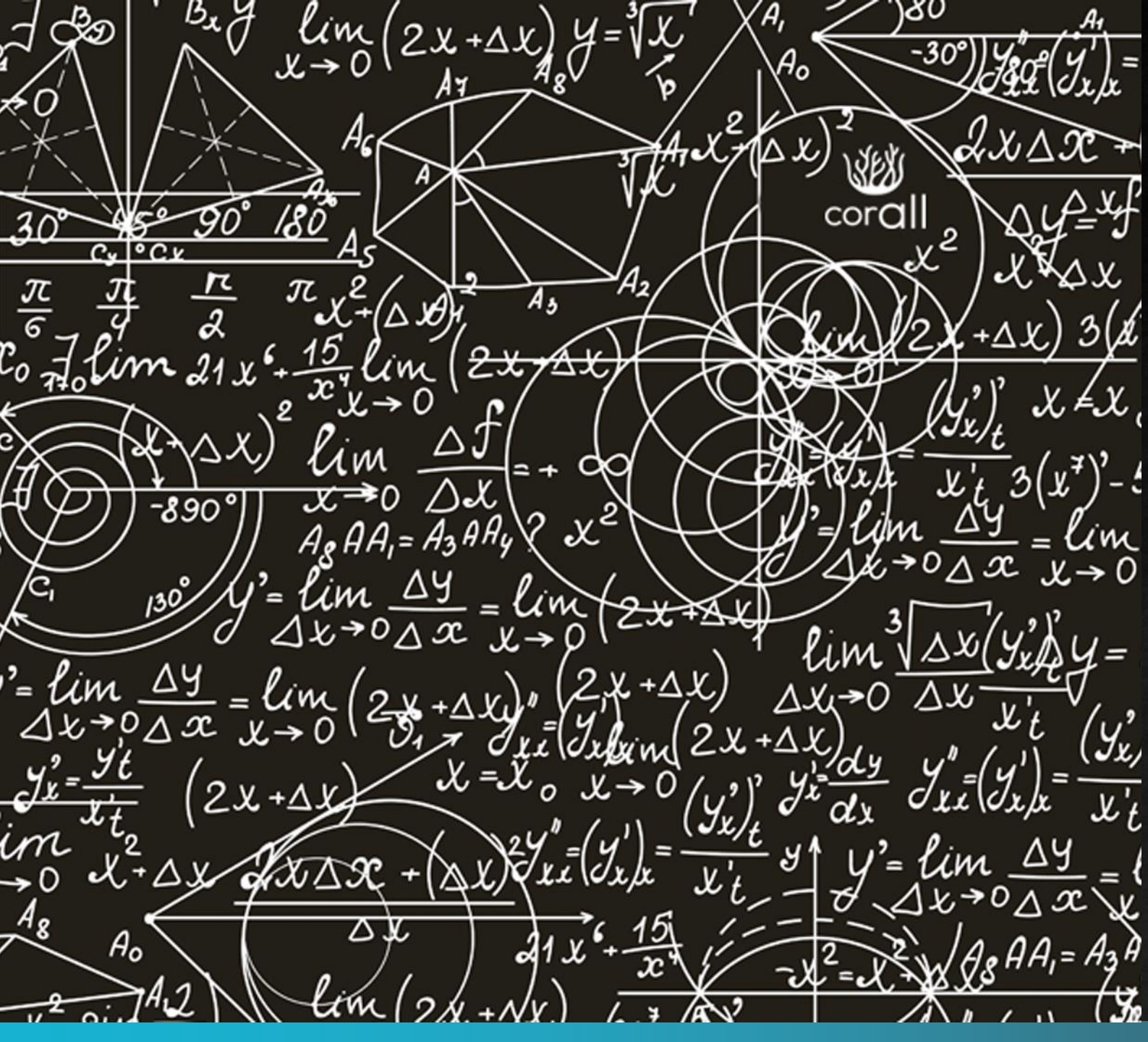


Noções de Complexidade Computacional

ED – 2021.2

Noções de Complexidade Algorítmica

- Objetivos
- Entender a importância da análise de algoritmos
- Entender e usar a notação Big-O
- Entender a implementação e avaliação de programas em linguagem Python

Noções de Complexidade Algorítmica

- **Questão:**
- Quando dois programas resolvem o mesmo problema, mas são diferentes, qual deles é o melhor?

- **Recordação:**
- Um algoritmo é genérico, é uma lista passo-a-passo de instruções para resolver um problema.
- Um programa, é um algoritmo que foi codificado em uma determinada linguagem de programação.

Noções de Complexidade Algorítmica

- Exemplo:
- Construa um programa que realize a soma dos **n** primeiros inteiros.

somaNinteiros.py

```python
# funcao soma_n - realiza o somatorio dos n primeiros inteiros
def soma_n(n):
    somatorio = 0
    for i in range(1, n+1):
        somatorio = somatorio + i
    return somatorio


print(soma_n(10))

```

Noções de Complexidade Algorítmica

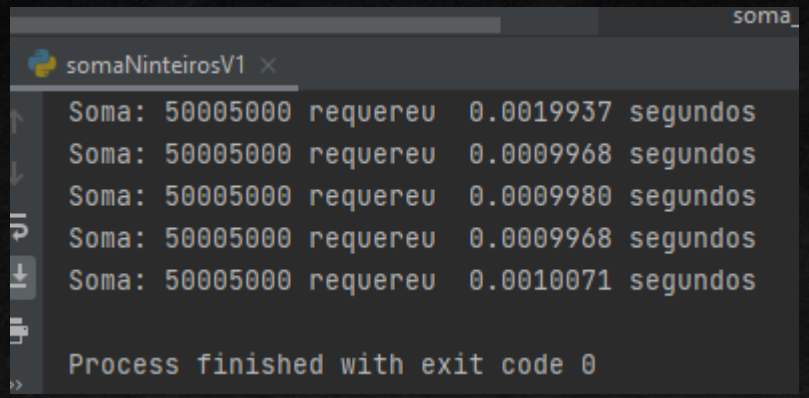
- A análise do algoritmo está preocupada em comparar algoritmos com base na quantidade de recursos de computação que cada algoritmo usa.
- É importante pensar mais sobre o que realmente queremos dizer com recursos de computação. Existem duas maneiras diferentes de olhar para isso.
  - Uma maneira é considerar a quantidade de espaço ou memória, um algoritmo requer para resolver o problema. A quantidade de espaço exigida por uma solução de problemas é tipicamente ditada pela própria instância do problema. De vez em quando, no entanto, existem algoritmos que têm equivalentes de espaço muito específicos, e neste caso, teremos muito cuidado para explicar as variações.
  - Como uma alternativa aos requisitos de espaço, podemos analisar e comparar algoritmos com base na quantidade de tempo que precisam ser executados. Esta medida é às vezes referida como o "tempo de execução" do algoritmo.

- Uma maneira de medir o tempo de execução para a função é fazer uma análise de benchmark. Isso significa que vamos acompanhar o tempo real necessário para o programa realizar as suas ações.
- Exemplo:

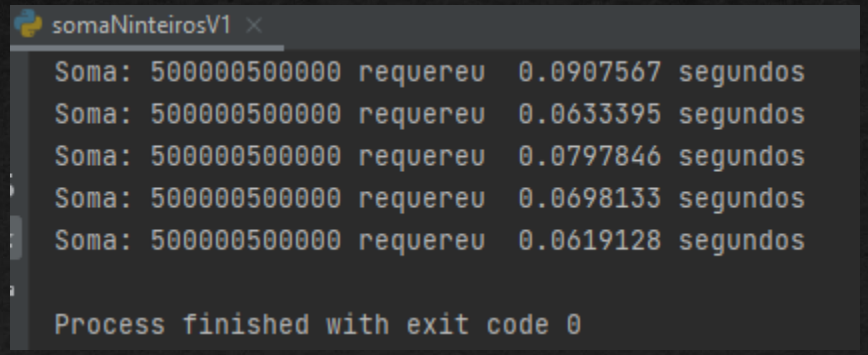
somaNinteiros.py | somaNinteirosV1.py

```python
import time


def soma_n_2(n):
    inicio = time.time()

    somatorio = 0
    for i in range(1, n+1):
        somatorio = somatorio + i

    fim = time.time()

    return somatorio, fim-inicio


for i in range(5):
    print("Soma: %d requereu %10.7f segundos" % soma_n_2(10000))

```

- Executando o programa para os primeiros 10.000 inteiros, teremos:



- Executando o programa para os primeiros 1.000.000 inteiros, teremos:



- Nesse exemplo, a média novamente é de cerca de 10 vezes mais segundos que o anterior, sendo bem consistente.
- No entanto, podemos ter uma outra forma para resolver o mesmo problema:

Ninteiros.py | somaNinteirosV1.py | somaNinteirosV2.py

```python
def soma_n_3(n):
    return (n * (n + 1)) / 2


print(soma_n_3(10))
```

- Fazendo a mesma medida de referência para soma_n_3, usando cinco valores diferentes para N (10.000, 100.000, 1.000.000, 10.000.000 e 100.000.000).
- Existem duas coisas importantes para perceber sobre essa saída:
  - Primeiro, os tempos gravados acima são mais curtos do que qualquer um dos exemplos anteriores.
  - Segundo, eles são muito consistentes, não importa qual o valor de ***n***. Parece que soma_n_3 dificilmente é impactado pelo número de inteiros adicionados.

- Mas existe um problema, pois se executarmos a mesma função em diferentes computadores ou usar diferentes linguagens de programação, teremos diferentes resultados.

Notação Big-O

- Ao tentar caracterizar a eficiência de um algoritmo em termos de tempo de execução, independente de qualquer programa ou computador específico, é importante quantificar o **número de operações ou etapas** que o algoritmo exigirá.
- Se cada uma dessas etapas for considerada uma unidade básica de computação, então o tempo de execução para um algoritmo pode ser expresso como o número de etapas necessárias para resolver o problema.
- Decidir sobre uma unidade básica apropriada de computação pode ser um problema complicado e dependerá de como o algoritmo é implementado.

Notação Big-O

- Exemplo: Uma boa unidade básica de computação para comparar os algoritmos de ***soma*** mostrados anteriormente pode ser contar o número de instruções de atribuição realizadas para calcular a soma.
  - Na função SOMA_N, o número de instruções de atribuição é 1 (somatório = 0) mais o valor de n (o número de vezes que realizamos somatório = somatório + i).
  - Podemos denotar isso por uma função, chamá-lo de T, onde T (n) = 1 + n. O parâmetro n é muitas vezes referido como o "tamanho do problema" e podemos ler isso como " T (n) é o tempo necessário para resolver um problema de tamanho n, nomeadamente 1 + n etapas."
- Nas funções de ***soma*** dadas, faz sentido usar o número de termos no somatório para denotar o tamanho do problema. Podemos dizer que a soma dos primeiros 100.000 inteiros é uma instância maior do problema de somatório do que a soma dos primeiros 1.000.
- Por causa disso, pode parecer razoável que o tempo necessário para resolver o caso maior ser maior do que para o caso menor. Nosso objetivo é então mostrar como o tempo de execução do algoritmo muda em relação ao tamanho do problema.

Notação Big-O

- Cientistas preferem tomar essa técnica de análise um passo adiante. Acontece que o número exato de operações não é tão importante quanto determinar a parte mais dominante da função $T(n)$.
- Em outras palavras, como o problema fica maior, alguma parte da função $T(n)$ tende a dominar o resto. Este termo dominante é o que, no final, é usado para comparação.
- A função de ordem de magnitude descreve a parte de $T(n)$ que aumenta o mais rápido que o valor de n aumenta. A ordem de magnitude é frequentemente chamada de notação **Big-O** e escrita como $O(f(n))$. Ele fornece uma aproximação útil para o número real de etapas na computação.
- A função $f(n)$ fornece uma representação simples da parte dominante do original $T(n)$.

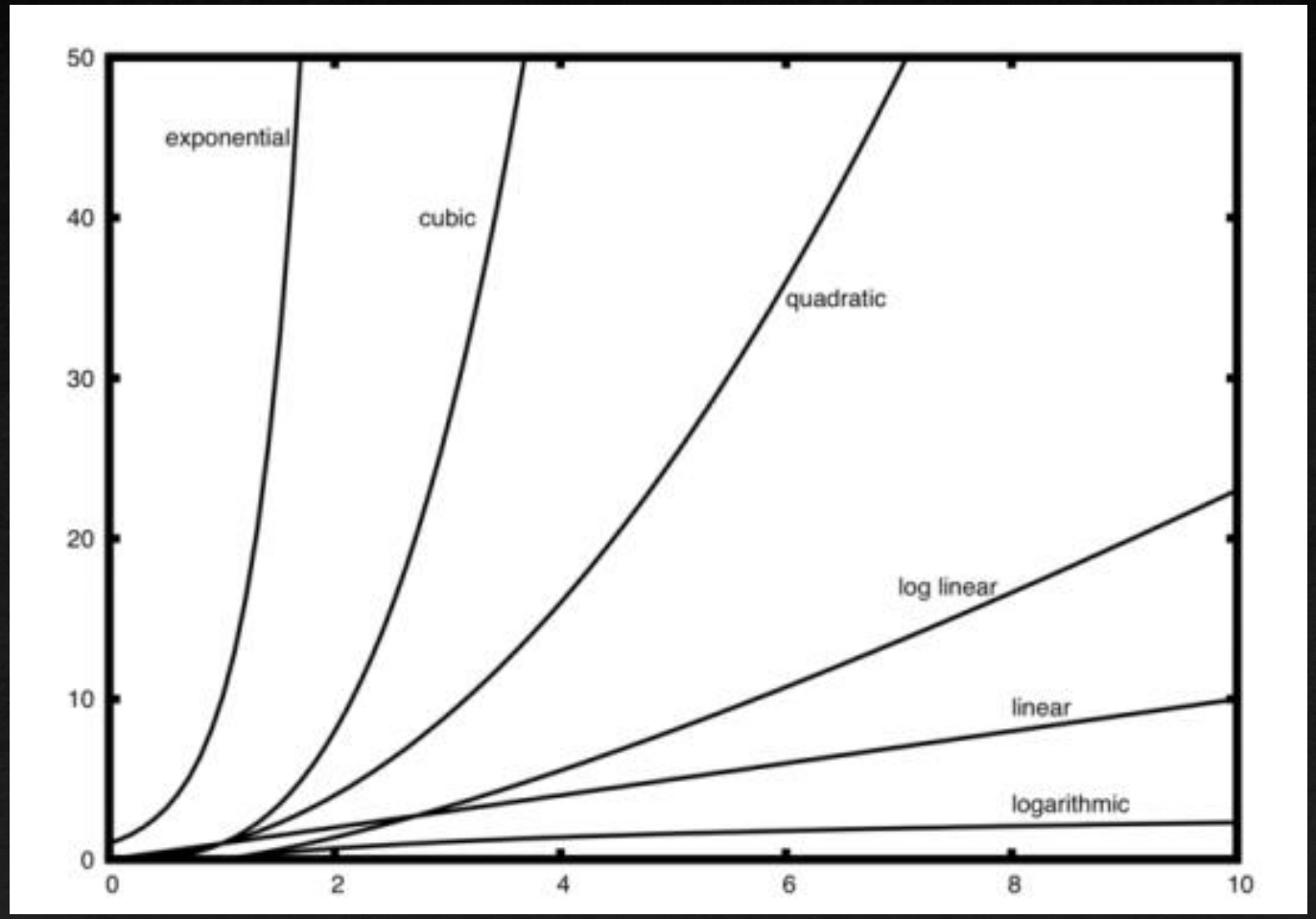
- No exemplo, temos $T\ (n) = 1 + n$. Como n fica grande, a constante 1 será cada vez menos significativa para o resultado final.
- Se estamos procurando uma aproximação para $T\ (n)$, então podemos soltar o 1 e simplesmente dizer que o tempo de execução é $O\ (n)$.
- É importante notar que o 1 é certamente significativo para $T\ (n)$. No entanto, como n fica grande, nossa aproximação será tão precisa sem ela.

- Outro exemplo: Suponha que, para algum algoritmo, o número exato de etapas é $T\ (n) = 5n^2 + 27n + 1005$.
  - Quando n é pequeno, digamos 1 ou 2, a constante 1005 parece ser a parte dominante da função.
  - No entanto, quando n fica maior, o termo n^2 torna-se o mais importante. Na verdade, quando n é realmente grande, os outros dois termos se tornarem insignificantes no papel que eles desempenham na determinação do resultado final.
  - Novamente, para aproximar $T\ (n)$ quando n é grande, podemos ignorar os outros termos e focar em $5n^2$. Além disso, o coeficiente 5 torna-se insignificante quando n é grande. Dizemos que a função $T\ (n)$ tem uma ordem de magnitude $f\ (n) = n^2$, ou simplesmente que é $O\ (n^2)$.

- Embora não vemos isso no exemplo de somatório, às vezes o desempenho de um algoritmo depende dos valores exatos dos dados, em vez de simplesmente o tamanho do problema.
- Para esses tipos de algoritmos, precisamos caracterizar seu desempenho em termos de melhor caso, pior caso ou desempenho médio do caso.
- O pior desempenho do caso refere-se a um conjunto de dados específico onde o algoritmo funciona especialmente mal. Considerando que um conjunto de dados diferente para o mesmo algoritmo pode ter um bom desempenho. No entanto, na maioria dos casos, o algoritmo realiza em algum lugar entre esses dois extremos (caso médio).
- É importante que o profissional de computação/informática compreenda essas distinções para que não sejam enganados por um caso particular.

Funções comuns para Big-O e gráfico

f(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential



- Como um exemplo final, suponha que temos o fragmento do código Python mostrado abaixo. Embora este programa não faça nada, é instrutivo ver como podemos fazer o código real e analisar o desempenho.

```python
# exemplo que não realiza nada

a = 5
b = 6
c = 10

for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j

for k in range(n):
    w = a * k + 45
    v = b * b
d = 33
```

- O número de operações de atribuição é a soma de quatro termos.
  - O primeiro termo é a constante 3, representando as três declarações de atribuição no início do fragmento.
  - O segundo termo é $3n^2$, uma vez que existem três declarações que são realizadas n^2 vezes devido a a iteração aninhada.
  - O terceiro termo é $2n$, duas declarações iteradas n vezes.
  - Finalmente, o quarto termo é a constante 1, representando a declaração final de atribuição. Isso nos dá $T\ (n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$.
  - Ao olhar para os expoentes, podemos ver facilmente que o termo n^2 será dominante e, portanto, este fragmento de código é $O\ (n^2)$. Observe que todos os outros termos, assim como o coeficiente no termo dominante, pode ser ignorado como cresce maior.

Desempenho de estruturas de dados Python

- Vamos contar sobre o desempenho do Big-O para as operações em listas e dicionários de Python e algumas experiências de tempo que ilustram os custos e benefícios de usar determinadas operações em cada estrutura de dados.
- É importante entender a eficiência dessas estruturas de dados Python, porque elas são os blocos de construção que usaremos para implementar outras estruturas de dados.
- Vamos ver algumas possíveis implementações de listas e dicionários e como o desempenho depende da implementação.

- **Listas**
- Os *designers* do Python tiveram muitas opções para fazer quando implementaram a estrutura de dados da lista.
- Cada uma dessas escolhas pode ter um impacto sobre como as operações de lista rápidas executam. Para ajudá-los a fazer as escolhas certas, eles analisaram os aspectos que as pessoas usariam mais a estrutura de dados da lista e otimizaram sua implementação para que as operações mais comuns fossem muito rápidas. É claro que eles (*designers*) também tentaram tornar as operações menos comuns, mas quando uma troca de uma operação menos comum era frequentemente sacrificada em favor da operação mais comum.

- Duas operações comuns estão indexando e atribuindo a uma posição de índice. Ambas as operações tomam a mesma quantidade de tempo, não importa quão grande seja a lista.
- Quando uma operação como esta é independente do tamanho da lista, eles são O (1).
- Outra tarefa de programação muito comum é crescer uma lista. Existem duas maneiras de criar uma lista mais longa. Você pode usar o método de anexação ou o operador de concatenação.
- O método de anexo é O (1). No entanto, o operador de concatenação é O (k) onde k é o tamanho da lista que está sendo concatenada.
- Isso é importante para você saber porque pode ajudá-lo a tornar seus próprios programas mais eficientes, escolhendo a ferramenta certa para o trabalho.

Operation	Big-O Efficiency
indexx[]	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

Eficiência Big-o dos operadores da lista em Python

- **Dicionário**
- A segunda estrutura de dados de python principal é o dicionário. Os dicionários diferem das listas porque você pode acessar itens em um dicionário por uma chave em vez de uma posição. O mais importante para notar agora é que o *"get item"* e definir operações de *item* em um dicionário são *O* (1). Outra operação importante do dicionário é a operação contém. Verificação para ver se uma chave está no dicionário ou não também é *O* (1).

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

FIM

Complexidade Algorítmica