Algoritmos Exponenciais

- Algoritmos exponenciais são geralmente impraticáveis de serem usados, mesmo com pequeno conjunto de dados de entrada.
- Funções recursivas que são chamadas repetidamente com os mesmos argumentos podem ser mais eficientemente utilizadas através de uma técnica chamada de ***memoization***.

Algoritmo Fatorial Recursivo

Fatorial:

1! = 1

2! = 2 * 1= 2

3! = 3 * 2 * 1 = 6

Função recursiva para calcular o fatorial de n:

```
def factorial(n):
    if n < 2:
        return 1
    elif n >= 2:
        return n * factorial(n-1)
```

Imprimir os 10 (dez) primeiros valores de fatorial:

```
for i in range(1, 11):
     print(f"{i}! = ", factorial(i))
```

```
1! =  1
2! =  2
3! =  6
4! =  24
5! =  120
6! =  720
7! =  5040
8! =  40320
9! =  362880
10! =  3628800
```

Tentativa de imprimir os 1.000 primeiros termos:

```
for i in range(1, 1000):
     print(f"{i}! = ", factorial(i))
```

```
1! =  1
2! =  2
3! =  6
4! =  24
5! =  120
6! =  720
7! =  5040
8! =  40320
9! =  362880
10! =  3628800
11! =  39916800
12! =  479001600
13! =  6227020800
14! =  87178291200
15! =  1307674368000
16! =  20922789888000
17! =  355687428096000
18! =  6402373705728000
19! =  121645100408832000
20! =  2432902008176640000
21! =  51090942171709440000
22! =  1124000727777607680000
23! =  25852016738884976640000
24! =  620448401733239439360000
25! =  15511210043330985984000000
26! =  403291461126605635584000000
27! =  10888869450418352160768000000
28! =  304888344611713860501504000000
29! =  8841761993739701954543616000000
30! =  265252859812191058636308480000000
31! =  8222838654177922817725562880000000
32! =  263130836933693530167218012160000000
33! =  8683317618811886495518194401280000000
34! =  295232799039604140847618609643520000000
35! =  10333147966386144929666651337523200000000
36! =  371993326789901217467999448150835200000000
37! =  13763753091226345046315979581580902400000000
38! =  523022617466601111760007224100074291200000000
39! =  20397882081197443358640281739902897356800000000
40! =  815915283247897734345611269596115894272000000000
41! =  33452526613163807108170062053440751665152000000000
42! =  1405006117752879898543142606244511569936384000000000
43! =  60415263063373835637355132068513997507264512000000000
44! =  2658271574788448768043625811014615890319638528000000000
45! =  119622220865480194561963161495657715064383733760000000000
46! =  5502622159812088949850305428800254892961651752960000000000
47! =  258623241511168180642964355153611979969197632389120000000000
48! =  12413915592536072670862289047373375038521486354677760000000000
49! =  608281864034267560872252163321295376887552831379210240000000000
50! =  30414093201713378043612608166064768844377641568960512000000000000
51! =  1551118753287382280224243016469303211063259720016986112000000000000
```

```
52! =  80658175170943878571660636856403766975289505440883277824000000000000
53! =  4274883284060025564298013753389399649690343788366813724672000000000000
54! =  230843697339241380472092742683027581083278564571807941132288000000000
55! =  12696403353658275925965100847566516959580321051449436762275840000000000
56! =  710998587804863451854045647463724949736497978881168458687447040000000
57! =  40526919504877216755680601905432322134980384796226602145184481280000
```

O erro `RecursionError: maximum recursion depth exceeded in comparison` indica que houve um estouro na pilha de memória (*stack overflow*). Isto ocorreu porque houve uma repetição de cálculos. Por exemplo, o fatorial de 2 foi calculados inúmeras vezes.

Cálculo do fatorial

Observe como a função recursiva calcula cada termo:

1! = 1

2! = 2 * 1= 2

3! = 3 * 2! = 6

4! = 4 * 3! = 24

Perceba que para calcular 4! nós estamos repetindo o cálculo de 3! e 2!.

Um maneira de se evitar essa repetição de cálculos é memorizar os cálculos previamente realizados.

Técnica de ***memoization***

- O programa mantém uma tabela de valores para cada argumento usado com a função.
- Antes da função recursivamente processar um valor para um dado argumento, ele checa a tabela para ver se o argumento já tem um valor.

Memoization para fatorial

Os valores previamente calculados serão armazenados em um dicionário.

```
fatorial_dict = {}
```

Em seguida, vamos definir a nossa função de memoization.

Inicialmente, nós verificamos se a entrada é menor que 2 e retornamos 1, caso seja

verdadeira a condição.

```
def fatorial_memo(n):
    if n < 2:
        return 1
```

Em seguida, nós checamos se o valor de entrada (n) está no dicionário. Se não estiver, nós armazenamos o valor no dicionário e retornamos o valor para a entrada n.

A função completa está descrita abaixo:

```
def fatorial_memo(n):
    if n < 2:
        return 1
    if n not in fatorial_dict:
        fatorial_dict[n] = n * fatorial_memo(n-1)
    return fatorial_dict[n]
```

Vamos imprimir os 1.000 primeiros números:

```
for i in range(1, 1000):
     print(f"{i}! = ", fatorial_memo(i))
```

```
1! =  1
2! =  2
3! =  6
4! =  24
5! =  120
6! =  720
7! =  5040
8! =  40320
9! =  362880
10! =  3628800
11! =  39916800
12! =  479001600
13! =  6227020800
14! =  87178291200
15! =  1307674368000
16! =  20922789888000
17! =  355687428096000
18! =  6402373705728000
19! =  121645100408832000
20! =  2432902008176640000
21! =  51090942171709440000
22! =  1124000727777607680000
23! =  25852016738884976640000
24! =  620448401733239439360000
25! =  15511210043330985984000000
26! =  403291461126605635584000000
27! =  10888869450418352160768000000
28! =  304888344611713860501504000000
```

```
29! =  8841761993739701954543616000000
30! =  265252859812191058636308480000000
31! =  8222838654177922817725562880000000
32! =  263130836933693530167218012160000000
33! =  8683317618811886495518194401280000000
34! =  295232799039604140847618609643520000000
35! =  10333147966386144929666651337523200000000
36! =  371993326789901217467999448150835200000000
37! =  13763753091226345046315979581580902400000000
38! =  523022617466601111760007224100074291200000000
39! =  20397882081197443358640281739902897356800000000
40! =  815915283247897734345611269596115894272000000000
41! =  33452526613163807108170062053440751665152000000000
42! =  1405006117752879898543142606244511569936384000000000
43! =  60415263063373835637355132068513997507264512000000000
44! =  2658271574788448768043625811014615890319638528000000000
45! =  119622220865480194561963161495657715064383733760000000000
46! =  5502622159812088949850305428800254892961651752960000000000
47! =  258623241511168180642964355153611979969197632389120000000000
48! =  12413915592536072670862289047373375038521486354677760000000000
49! =  608281864034267560872252163321295376887552831379210240000000000
50! =  30414093201713378043612608166064768844377641568960512000000000000
51! =  1551118753287382280224243016469303211063259720016986112000000000000
52! =  80658175170943878571660636856403766975289505440883277824000000000000
53! =  4274883284060025564298013753389399649690343788366813724672000000000000
54! =  230843697339241380472092742683027581083278564571807941132288000000000
55! =  12696403353658275925965100847566516959580321051449436762275840000000000
56! =  710998587804863451854045647463724949736497978881168458687447040000000
57! =  40526919504877216755680601905432322134980384796226602145184481280000
58! =  2350561331282878571829474910515074683828862318181142924420699914240
```

Decorators em Python

Em Python, a técnica de *memoization* pode ser executada com a ajuda de funções *decorators* de Python.

Python fornece uma biblioteca que realiza automaticamente essa operação: `lru_cache`.

```
from functools import lru_cache

@lru_cache(maxsize = 1000)
def fatorial(n):
    if n < 2:
        return 1
    elif n >= 2:
        return n * fatorial(n-1)

for i in range(1, 1000):
     print(f"{i}! = ", fatorial(i))
```

```
1! =  1
2! =  2
3! =  6
```

```
4! =  24
5! =  120
6! =  720
7! =  5040
8! =  40320
9! =  362880
10! =  3628800
11! =  39916800
12! =  479001600
13! =  6227020800
14! =  87178291200
15! =  1307674368000
16! =  20922789888000
17! =  355687428096000
18! =  6402373705728000
19! =  121645100408832000
20! =  2432902008176640000
21! =  51090942171709440000
22! =  1124000727777607680000
23! =  25852016738884976640000
24! =  620448401733239439360000
25! =  15511210043330985984000000
26! =  403291461126605635584000000
27! =  10888869450418352160768000000
28! =  304888344611713860501504000000
29! =  8841761993739701954543616000000
30! =  265252859812191058636308480000000
31! =  8222838654177922817725562880000000
32! =  263130836933693530167218012160000000
33! =  8683317618811886495518194401280000000
34! =  295232799039604140847618609643520000000
35! =  10333147966386144929666651337523200000000
36! =  371993326789901217467999448150835200000000
37! =  13763753091226345046315979581580902400000000
38! =  523022617466601111760007224100074291200000000
39! =  20397882081197443358640281739902897356800000000
40! =  815915283247897734345611269596115894272000000000
41! =  33452526613163807108170062053440751665152000000000
42! =  1405006117752879898543142606244511569936384000000000
43! =  60415263063373835637355132068513997507264512000000000
44! =  2658271574788448768043625811014615890319638528000000000
45! =  119622220865480194561963161495657715064383733760000000000
46! =  5502622159812088949850305428800254892961651752960000000000
47! =  258623241511168180642964355153611979969197632389120000000000
48! =  12413915592536072670862289047373375038521486354677760000000000
49! =  608281864034267560872252163321295376887552831379210240000000000
50! =  30414093201713378043612608166064768844377641568960512000000000000
51! =  1551118753287382280224243016469303211063259720016986112000000000000
52! =  80658175170943878571660636856403766975289505440883277824000000000000
53! =  4274883284060025564298013753389399649690343788366813724672000000000000
54! =  230843697339241380472092742683027581083278564571807941132288000000000
55! =  12696403353658275925965100847566516959580321051449436762275840000000
56! =  710998587804863451854045647463724949736497978881168458687447040000000
57! =  40526919504877216755680601905432322134980384796226602145184481280000
58! =  2350561331282878571829474910515074683828862318181142924420699914240 0
```

O valor da memória cache para o decorator de Python é, por padrão, sem limites. Adotar essa liberalidade pode não ser uma boa ideia, pois o programa pode utilizar toda a memória

disponível no computador e que deveria ser usada também por outros programas.

Exercícios

1. Implemente uma função recursiva para calcular a série de Fibonacci. Utilize a técnica de *memoization*.
2. A função de *memoization* só faz sentido se ela for determinística. Você concorda com essa afirmação? Justifique a sua resposta.
3. A função de memoize criada por um programador tende a ser mais rápida do que a função `lru_cache` fornecida pelo Python padrão? Justifique a sua resposta.
4. Escreva um código em Python para mostrar o gráfico comparativo de tempo de execução com cada estratégia apresentada acima.

Resposta exercício 4.

```
import time

N = 50

#Função sem memoization
def factorial(n):
    if n < 2:
        return 1
    elif n >= 2:
        return n * factorial(n-1)

#tempo metodo1
tempo1 = []
for i in range(1, N):
  start_time = time.perf_counter ()
  end_time = time.perf_counter ()
  tempo1.append(end_time - start_time)


#Função memoization criada pelo usuário
fatorial_dict = {}

def fatorial_memo(n):
    if n < 2:
        return 1
    if n not in fatorial_dict:
        fatorial_dict[n] = n * fatorial_memo(n-1)
    return fatorial_dict[n]

#tempo metodo2
tempo2 = []
```

```
for i in range(1, N):
  start_time = time.perf_counter ()
  end_time = time.perf_counter ()
  tempo2.append(end_time - start_time)

from functools import lru_cache

#Função memoization padrão Python
@lru_cache(maxsize = 100)
def fatorial(n):
    if n < 2:
        return 1
    elif n >= 2:
        return n * fatorial(n-1)

#tempo metodo3
tempo3 = []
for i in range(1, N):
  start_time = time.perf_counter ()
  end_time = time.perf_counter ()
  tempo3.append(end_time - start_time)

print (tempo1)
print (tempo2)
print (tempo3)
```

```
[7.060000086767104e-07, 3.179999907843012e-07, 2.0300001324358163e-07, 2.11
[7.229999994251557e-07, 2.8600001655831875e-07, 2.599999788799323e-07, 2.44
[3.9799999740353087e-07, 2.660000006926566e-07, 2.4000001985768904e-07, 2.3
```

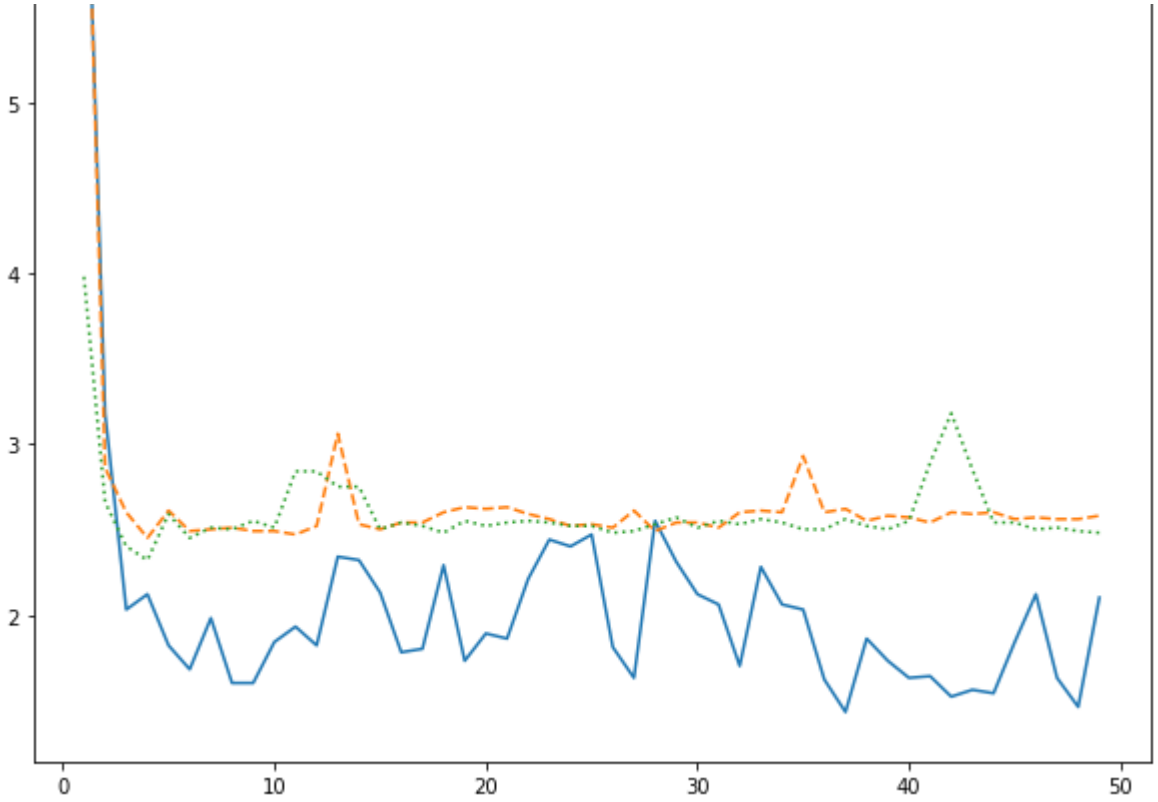
```
import matplotlib.pyplot as plt
import numpy as np

y = np.arange(1,N)

plt.figure(figsize=(10, 10))

plt.plot(y, tempo1, label = "Método 1", linestyle="-")
plt.plot(y, tempo2, label = "Método 2", linestyle="--")
plt.plot(y, tempo3, label = "Método 3", linestyle=":")

plt.legend()
plt.show()
```


5
4
3
2
0
10
20
30
40
50