Análise de Algoritmos

Conceitos básicos

Análise de Algoritmos

- Objetivo: identificar algoritmos mais **eficientes**.
  - Pode existir mais de um candidato viável.
  - Descartamos algoritmos inferiores durante o processo de análise.

Análise de Algoritmos

- Diferentes algoritmos que solucionam um mesmo problema podem apresentar uma diferença significativa em termos de eficiência.
- Tais diferenças podem ser mais impactantes que diferenças de hardware e software.

Problema Computacional: Ordenação

- Problema computacional que surge em diversas situações.
- Definição:
  - Input: Uma sequência de n números $\{a_1\ a_2, a_3, \ldots, a_n\}$
  - Output: Uma reordenação da sequência em $\{a'_1, a'_2, a'_3 \ldots a'_n\}$ tal que $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$

Problema Computacional: Ordenação

- Exemplo: Ordenar 10^7 números.
  - Suponha que o computador A execute 10 bilhões de tarefas por segundos.
  - Um excelente programador implementa o algoritmo X em linguagem de máquina no computador A conseguindo performance da ordem de n^2, onde n é o tamanho da entrada (10^7).

Problema Computacional: Ordenação

- Exemplo: Ordenar 10^7 números.
  - Suponha que o computador B execute 10 milhões de tarefas por segundos.
  - Um programador com nível mediano implementa o algoritmo B usando Python. Ele consegue uma performance proporcional a nlogn, onde n é o tamanho da entrada.
- **Lembre-se: o computador A é 1000 vezes mais rápido do que o computador B.**

Problema Computacional: Ordenação

- Exemplo: Ordenar 10^7 números.

$$A = \frac{(10^7)^2}{10^{10}} = 10^4 \approx 2.7h$$

$$B = \frac{10^7 log 10^7}{10^7} = 7s$$

Eficiência

- A eficiência não recai sobre o tipo de linguagem utilizada para implementar o algoritmo, nem sobre o tipo de máquina na qual o algoritmo é executado.
- A eficiência precisa ser avaliada através de um critério adequado para comparar algoritmos.
- Esse critério deve ser independente da linguagem e tipo de máquina.

Eficiência

- Tempo de execução de um programa.
  - Pode variar de acordo com o algoritmo.
  - Pode variar para um mesmo algoritmo implementado de diferentes formas.
  - Pode variar para uma mesma implementação executada em computadores diferentes.
  - A estimativa de tempo não pode ser baseada em entradas de tamanho reduzido.

Eficiência

- Contagem das operações executadas.
    - Depende do algoritmo.
    - Depende do tipo de implementação.
    - Não depende do tipo de máquina.
    - Nem sempre são claras quais operações serão contadas.

Eficiência

```
def soma1(n):
    soma = 0
    for i in range(n):
        soma +=i
    return soma
```

1op: atrib
2op: atrib + soma - n vezes
2op: atrib + soma - n vezes
1op: retorno

Total = 1+2n+2n+1=4n+2

Eficiência

```
def soma2(n):
    soma = 0
    i=0
    while i< n:
        soma +=i
        i+=1
    return soma
```

1op: atrib
1op: atrib
1op: comparação - n vezes
2op: atrib + soma - n vezes
2op: atrib + soma - n vezes
1op: retorno

Total = 1+1+n+2n+2n+1 = 5n+3

```
while i< n:
        soma +=i
        i+=1
n=3
0< n:
soma +=0
i+=1

1< n:
soma +=1
i+=1

2<n
soma += 2
i+=1
```

Eficiência

- Temos:
  soma1(): 4n+2
  soma2(): 5n+3
- Tanto soma1() quanto soma 2() apresentam ordem de crescimento linear no tempo de execução em função do tamanho da entrada n.

Eficiência

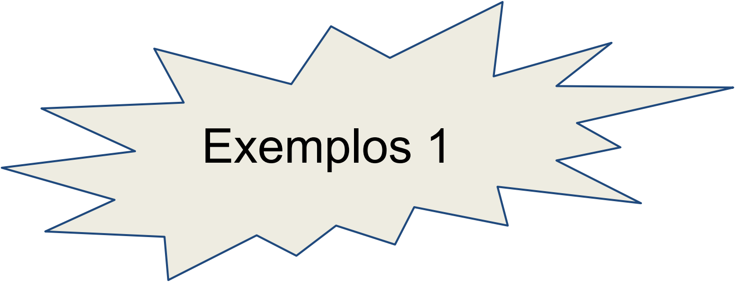
- **Melhor caso:** tempo mínimo de execução para todas as entradas possíveis de um dado tamanho n.

- **Caso médio:** tempo médio de execução para todas as entradas possíveis de um dado tamanho n, ou estimativa sobre o tempo de execução.

- **Pior caso:** tempo máximo de execução para todas as entradas possíveis de um dado tamanho n.

Eficiência

- Nesse contexto, a **eficiência** de um algoritmo pode ser definida como sua **complexidade de tempo**.
- A **complexidade de tempo** é dada pelo número de **instruções básicas** que ele executa considerando o tamanho da entrada.
- Observa-se que o **pior caso** possível é considerado na determinação do tempo máximo para solucionar uma instância de grande porte.
- Trata-se de uma **análise assintótica**, ou seja, avalia-se o comportamento do algoritmo para entradas de grande porte.


Exemplos 1

RAM Model

- Random-access machine (RAM) model
  - Permite uma adequada comparação entre algoritmos diferentes aplicados a um mesmo problema.
  - Facilita a avaliação do comportamento dos algoritmos em instâncias de grande porte.

RAM Model

- Random-access machine (RAM) model
  - Sequencial:
    - A instruções são executadas uma após a outra , ou seja, sem operações simultâneas.
  - Há um único processador.

RAM Model

- Random-access machine (RAM) model
  - Tipos de instruções consideradas:
    - aritméticas (adicionar, subtrair, multiplicar, dividir, resto, piso, teto)
    - transferências de dados (carregar, armazenar, copiar)
    - estruturas de controle.
  - Cada instrução consome uma quantidade de tempo constante.

RAM Model

- Random-access machine (RAM) model
  - Tipos básicos: inteiros e reais.
  - Um limite no tamanho de cada palavra de dados é assumido.
    - Por exemplo, assumimos que inteiros são representados por $c\lg n$ bits para alguma constante $c \geq 1$ e entradas de tamanho n.

RAM Model

- Desvantagens
  - Análise focada no pior caso que pode não ocorrer frequentemente.
  - Não permite análise do comportamento no caso médio

Ordem de Crescimento

- Permite avaliar a eficiência do programa considerando entradas grandes.
- Estima-se o crescimento do tempo de execução do programa quando o tamanho da entrada cresce.
- Não precisa avaliar o crescimento exato, mas sim a ordem do crescimento.

Ordem de Crescimento

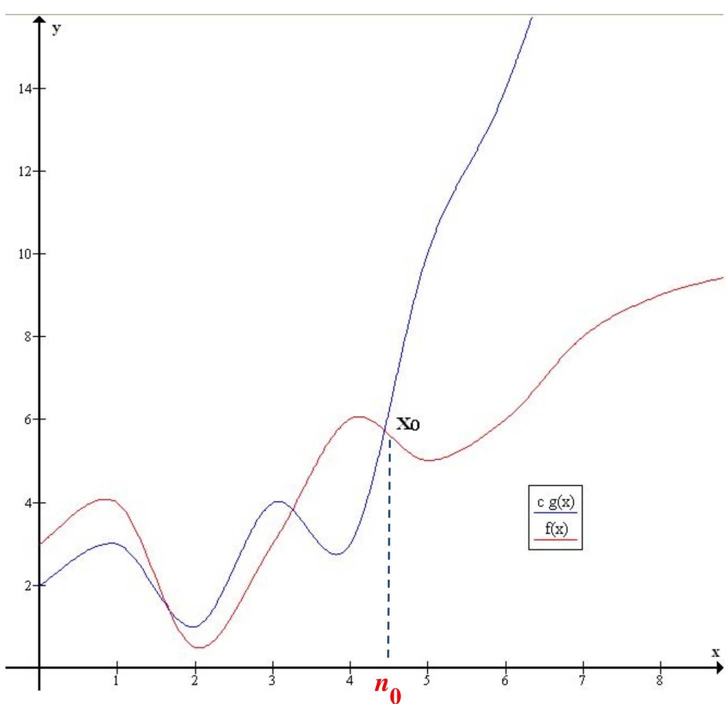
- Na maioria dos casos, deseja-se obter um **limitante superior** para o **crescimento do tempo de execução** em função do **tamanho da entrada** no **pior caso**.

- Utiliza-se a notação big Oh ou O() para obter um limitante superior na ordem de crescimeno.

- O() é utilizado para avaliar o pior caso.

Ordem de Crescimento

$$O(\boldsymbol{g}(\boldsymbol{n})) = \{ f(n) : \exists\, c > 0, n_0 > 0 \text{ tais que } 0 \leq f(n) \leq c \cdot g(n)\ \forall\, n \geq n_0 \}$$



Ordem de Crescimento

```
def fat(n):
   f=1                        1op: atribuição
   for i in range(1,n+1):     2op: atribuição+soma   n vezes
      f=f*i                   2op: atribuição+produto n vezes
   return f                   1op: retorno
```

$f(n) = 4n+2$

$O(f(n))=O(4n+2)=$ **$O(n)$** ⇒ crescimento de ordem **linear**

Ordem de Crescimento

```
def buscarElemento(L, x):
    for i in L:
        if i == x:
            return True
    return False
```

2op: atribuição+soma, len(L) vezes
1op: comparação, len(L) vezes
1op: retorno, 1 vez
1op: retorno, 1 vez

$f(n) = 3n+2$

$O(f(n)=O(3n+2)=$**$O(n)$** $\Rightarrow$ crescimento de ordem **linear no pior caso**

Ordem de Crescimento

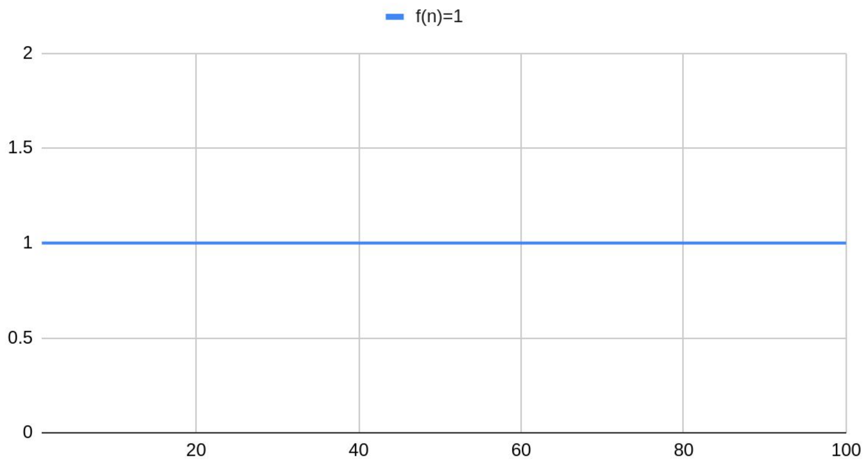
```
def pot2(n):
    pot2 = 0                  1op: atribuição
    for i in range(n):        2op: atribuição+soma, n vezes
        for j in range(n):    2op: atribuição+soma, n² vezes
            pot2 += 1         2op: atribuição+soma, n² vezes
    return pot2               1op: retorno
```

$f(n) = 2n^2+2n+2$

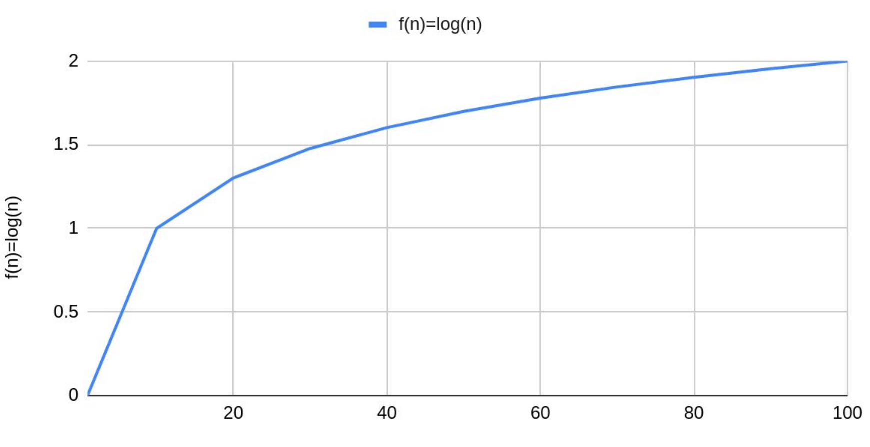
$O(f(n))=O(n^2)$

Ordem de Crescimento

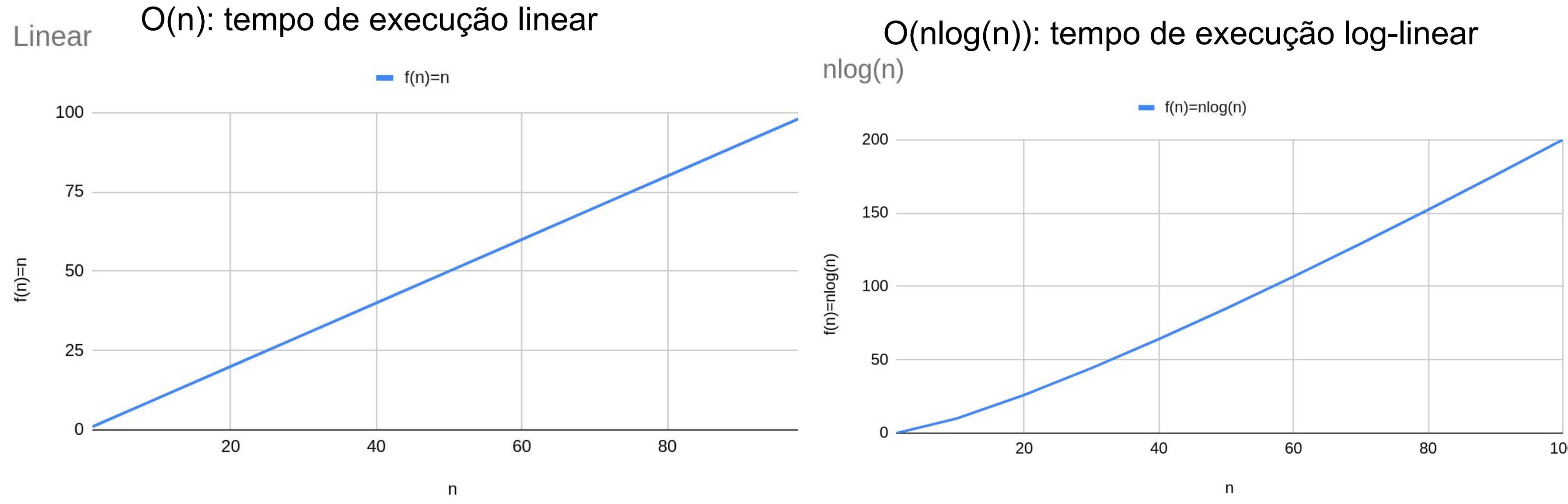
O(1): Tempo de execução constante



O(log(n)): Tempo de execução logarítmico

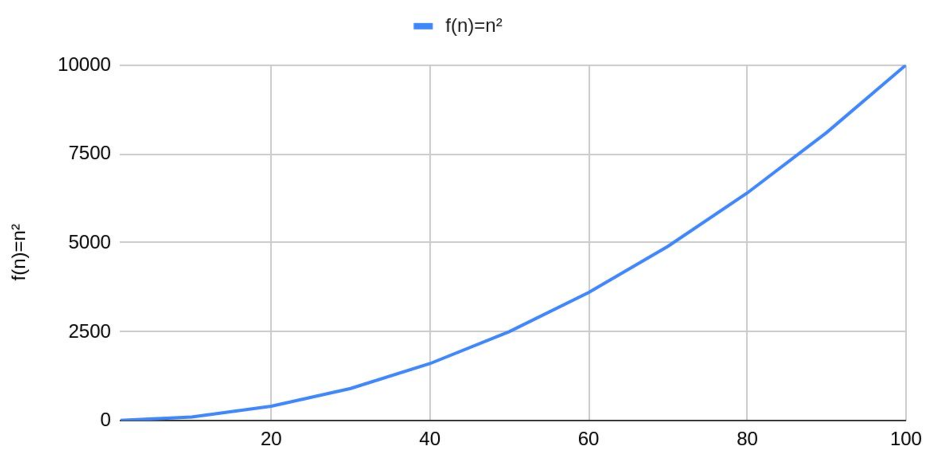


Ordem de Crescimento

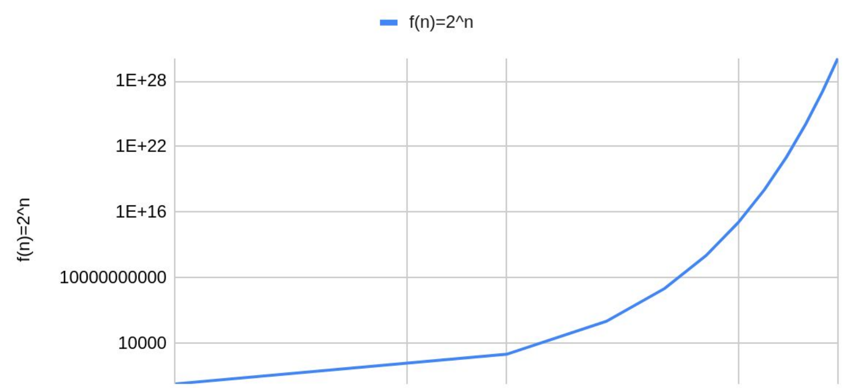


Ordem de Crescimento

$O(n^C)$: Tempo de execução polinomial



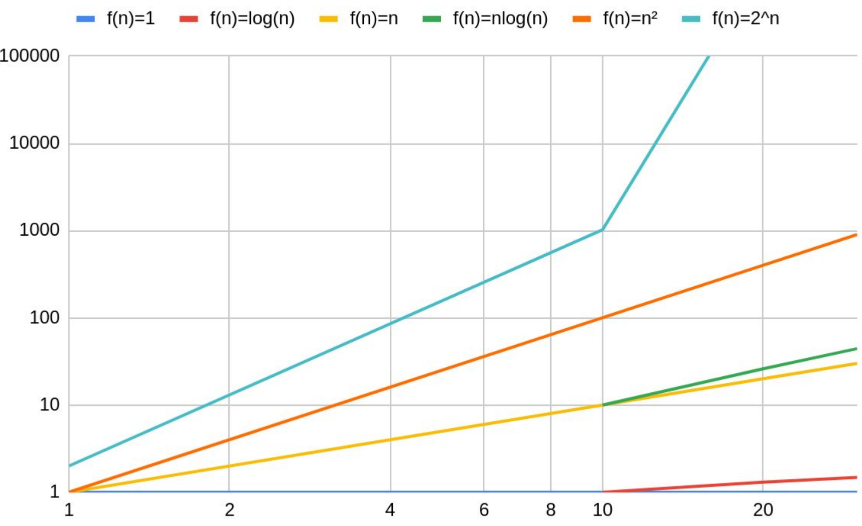
$O(c^n)$: Tempo de execução exponencial



c constante

Ordem de Crescimento

n	f(n)=1	f(n)=log(n)	f(n)=n	f(n)=nlog(n)	f(n)=n²	f(n)=2^n
1	1	0	1	0	1	2
10	1	1	10	10	100	1024
20	1	1.301029996	20	26.02059991	400	1048576
30	1	1.477121255	30	44.31363764	900	1073741824



Ordem de Crescimento

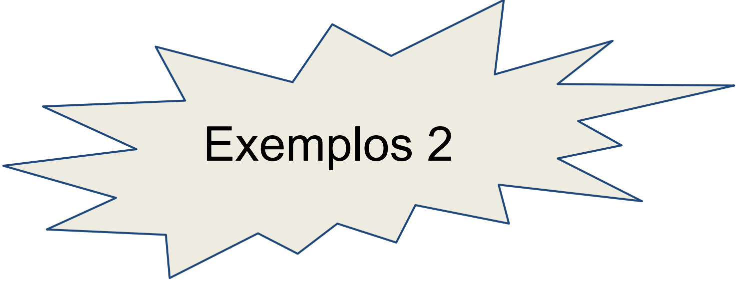
Resumindo, podemos aplicar as regras abaixo na definição da complexidade assintótica de um algoritmo:

- Se o tempo de execução for a soma de vários termos, mantenha aquele com a maior taxa de crescimento e descarte os outros.

- Se o termo restante for um produto, elimine quaisquer constantes.

Resumindo

- O(1): código não depende do tamanho da entrada.
- O(logn): código reduz o tamanho da entrada pela metade.
- O(n): códigos iterativos ou recursivos.
- O(nlogn): códigos de alguns algoritmos de ordenação.
- $O(n^c)$: laços chamadas recursivas aninhados.
- $O(c^n)$: múltiplas chamadas recursivas.


Exemplos 2

Fatorial

```
def fatIter(n):
    fat=1
    for i in range(1,n+1):
        fat *= i
    return fat
```

O(n)

```
def fatRec(n):
    if n<2:
        return 1
    else
        return n*fatRec(n-1)
```

O(n)

Converter inteiro para string

```
def converteInt_Str(x):
    inteiros_str = '0123456789'
    if x == 0:
        return '0'
    x_str = ''
    while x > 0:
        x_str = inteiros_str[x%10] + x_str
        x = x//10
    return x_str
```

O(logx)

Converter inteiro para string

```
1  def subConjunto(L1, L2):
2      '''
3      Entrada: L1, L2
4      Saída: True or False, Se L1 eh subconjunto de L2
5      '''
6      for i in L1:
7          estaContido = False
8          for j in L2:
9              if i == j:
10                 estaContido = True
11                 break
12         if not estaContido:
13             return False
14     return True
```

O(len(L1))

O(len(L2))

O(len(L1)*len(L2))

$O(n*n)=O(n^2)$

Converter inteiro para string

```
1  def intersec(L1, L2):
2      """Entrada: L1 e L2
3          Saída: Lista com elementos em comum"""
4      #Armazena elementos em comum
5      tmp = []
6      for i in L1:
7          for j in L2:
8              if i == j:
9                  tmp.append(i)
10     #Remove duplicatas
11     intersec = []
12     for i in tmp:
13         if i not in intersec:
14             intersec.append(i)
15     return intersec
```

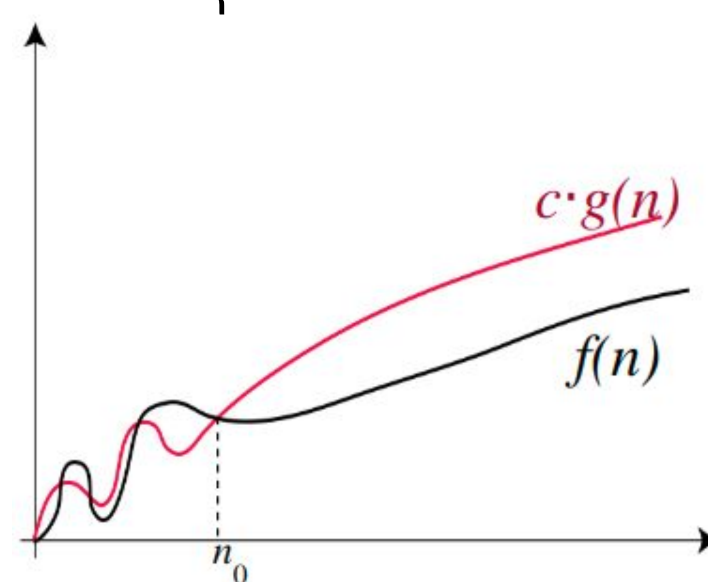
$O(n^2)$

$O(len(L1)*len(L2))$
$O(n^2)$

$O(len(tmp)*len(intersec))$
$O(n^2)$

Notação - O

$O(\boldsymbol{g}(\boldsymbol{n})) = \{ f(n) : \exists\, c > 0, n_0 > 0 \text{ tais que } 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0 \}$



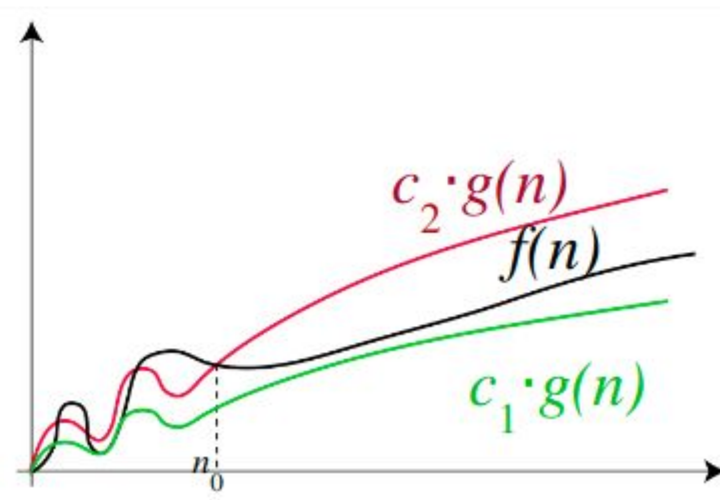
Notação - *O*

- Observe que $O(n^2)$ significa que há uma função n que é $O(n^2)$ tal que para qualquer valor de n, não importa o tamanho específico de entrada n escolhido, o tempo de execução daquela entrada é $O(n^2)$.

- Isso significa que o tempo de execução no pior caso é $O(n^2)$.

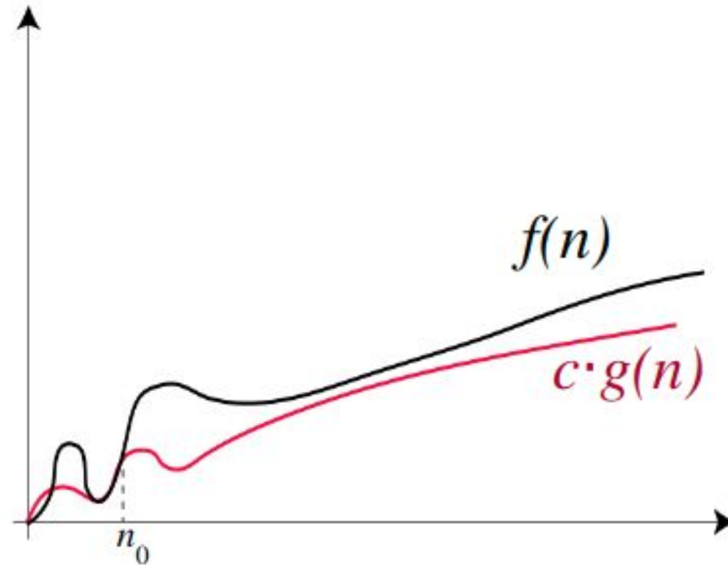
Notação - Θ

$$\Theta(g(n)) = \{\, f(n) : \exists\, c_1, c_2 > 0, n_0 > 0 \text{ tais que } 0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)\ \forall n \ge n_0 \,\}$$



Notação - Ω

$$\Omega(\boldsymbol{g(n)}) = \{\, f(n) : \exists\, c > 0, n_0 > 0 \text{ tal que } 0 \le c \cdot g(n) \le f(n) \;\; \forall n \ge n_0 \,\}$$



Complexidade manipulando listas em Python

Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Pop last	O(1)	O(1)
Pop intermediate[2]	O(n)	O(n)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

Complexidade manipulando dicionários em Python

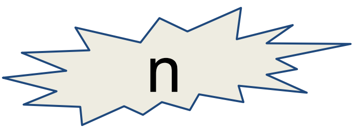
k in d	O(1)	O(n)
Copy[3]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[3]	O(n)	O(n)

```python
def search(L, e):
    """Assumes L is a list, the elements of which are in
          ascending order.
       Returns True if e is in L and False otherwise"""

    def bSearch(L, e, low, high):
        #Decrements high - low
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bSearch(L, e, low, mid - 1)
        else:
            return bSearch(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return bSearch(L, e, 0, len(L) - 1)
```

Busca Binária



Mínimo Médio Máximo

Mínimo Médio Máximo

n/2

n/4

1

Mínimo Médio Máximo

...

Mínimo Máximo

$n/2^i$

Busca Binária

- Quando a Busca termina?

$$n/2^i=1 \Rightarrow n=2^i \Rightarrow \log_2 n=i$$

- Logo, a complexidade do algoritmo recursivo é $O(\log_2 n)$.