Fundamentos da Programação

Algoritmos de Procura e de Ordenação

Aula 17

Alberto Abad, IST, 2022-23

Algoritmos de Procura

- A procura de um elemento numa **lista** é uma das operações mais comuns sobre listas.
- O objetivo do processo de procura em uma lista `l` é descobrir se o valor `x` está na lista e em que posição.
- Existem múltiplos algoritmos de procura (alguns mais eficientes e outros menos).
- Hoje vamos ver:
    - Procura sequencial ou linear
    - Procura binária

Algoritmos de Procura - Procura Sequencial

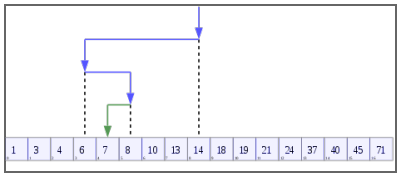
```
In [6]: ## Procurar indice de primeira ocurrencia de x, ou devolver -
        def linearsearch(l, x):
    for i in range(len(l)):
        if l[i] == x:
            return i
    return -1


%timeit -n 1000 linearsearch(list(range(1000)), 700)
%timeit -n 1000 (list(range(1000)).index(700))
```

```
28.4 µs ± 8.91 µs per loop (mean ± std. dev. of 7 runs,
1,000 loops each)
11.1 µs ± 338 ns per loop (mean ± std. dev. of 7 runs, 1
,000 loops each)
```

- O número de comparações depende da posição onde se encontrar o elemento, pode ir de `1` até `n` se o elemento não se encontrar na lista.
- Será que conseguimos fazer melhor?

Algoritmos de Procura - Procura Binária

- Podemos fazer melhor se a lista estiver ordenada!!




```
In [21]: from math import log2

def binsearch(l, x):
    inf = 0
    sup = len(l) -1

    while inf <= sup:
        m = inf + (sup - inf)//2
        if l[m] == x:
            return m
        elif l[m] > x:
            sup = m - 1
        else:
            inf = m + 1

    return -1


from random import shuffle
l = list(range(1000))
r = l.copy()
shuffle(r)

%timeit -n 1000 linearsearch(r, 800)
%timeit -n 1000 binsearch(l, 800)


log2(10000)
```

```
3.94 µs ± 904 ns per loop (mean ± std. dev. of 7 runs, 1
,000 loops each)
1.58 µs ± 111 ns per loop (mean ± std. dev. of 7 runs, 1
```

```
,000 loops each)
Out[21]:
13.287712379549449
```

Algoritmos de Ordenação

- Isto não significa que seja sempre melhor ordenar e procurar depois.
- Em geral, a ordenação têm um custo superior que a procura linear, e manter uma lista ordenada também é custoso.
- No entanto, se o número de procuras for muito superior ao número de alterações na lista, compensa ordenar e utilizar a pesquisa binária.
- Existem vários algoritmos de **ordenação** e, em Python, temos as funções pré-definidas `sorted` e a função `sort` sobre listas, que implementa um desses algoritmos de ordenação chamado *Timsort*.

```
>>> l = [1,8,21,4,1,8,9]
>>> sorted(l)
[1, 1, 4, 8, 8, 9, 21]
>>> l
[1, 8, 21, 4, 1, 8, 9]
>>> l.sort()
>>> l
[1, 1, 4, 8, 8, 9, 21]
>>>
```

```
In [88]: l = [1,8,21,4,1,8,9]
         l2 = sorted(l)
print(l)
print(l2)
```

```
(1, 8, 21, 4, 1, 8, 9)
[1, 1, 4, 8, 8, 9, 21]
```

Algoritmos de Ordenação - *Bubble sort*

https://visualgo.net/pt/sorting

In [26]:
```
from random import shuffle
nums = list(range(1000))
shuffle(nums)

# print(nums)

def bubblesort(l):
    for i in range(len(l)):
        changed = False
        for j in range(len(l)-1-i):
            if l[j] > l[j+1]:
                l[j], l[j+1] =  l[j+1], l[j]
                changed = True

        if not changed:
            break


nums1=nums.copy()
%time bubblesort(nums1)
# print(nums1)
```

```
CPU times: user 68.5 ms, sys: 2.05 ms, total: 70.6 ms
Wall time: 69.3 ms
```

Algoritmos de Ordenação - *Shell Sort*

In [30]:
```
def bubblesort(l, step = 1):
            changed = True
    size = len(l) - step
    while changed:
        changed = False
        for i in range(size): #maiores para o fim da lista
            if l[i] > l[i+step]:
                l[i], l[i+step] = l[i+step], l[i]
                changed = True
        size = size -1

def shellsort(l):
    step = len(l)//2
    while step != 0:
        bubblesort(l, step)
        step = step//2


nums = list(range(1000))
shuffle(nums)

nums1=nums.copy()
%time bubblesort(nums1)
print(nums1 == sorted(nums1))

nums2=nums.copy()
%time shellsort(nums2)
print(nums2 == sorted(nums2))
```

```
CPU times: user 66.1 ms, sys: 2.47 ms, total: 68.6 ms
Wall time: 67.1 ms
True
CPU times: user 3.92 ms, sys: 6 µs, total: 3.92 ms
Wall time: 3.93 ms
True
```

Algoritmos de Ordenação - *Selection Sort*

In [29]:
```
def selectionsort(lista):
        for i in range(len(lista)):
        minimum = i
        for j in range(i+1, len(lista)):
            if lista[j] < lista[minimum]:
                minimum = j
        lista[i], lista[minimum] = lista[minimum], lista[i]



nums3 = nums.copy()
%time selectionsort(nums3)
print(nums3 == sorted(nums3))
```

```
CPU times: user 36.5 ms, sys: 2.57 ms, total: 39.1 ms
Wall time: 37.4 ms
True
```

Algoritmos de Ordenação - *Insertion Sort*

```
In [33]: def insertionsort(l):
             for i in range(1, len(l)):
         x = l[i]
         j = i - 1
         while j >= 0 and x < l[j]:
             l[j+1] = l[j]
             j = j - 1
         l[j+1] = x


def insertionsort(l):
    for i in range(1, len(l)):
        x = l[i]
        for j in range(i-1,-1,-1):
            if x < l[j]:
                l[j+1] = l[j]
            else:
                l[j+1] = x
                break



nums4=nums.copy()
%time insertionsort(nums4)
print(nums4 == sorted(nums4))
```

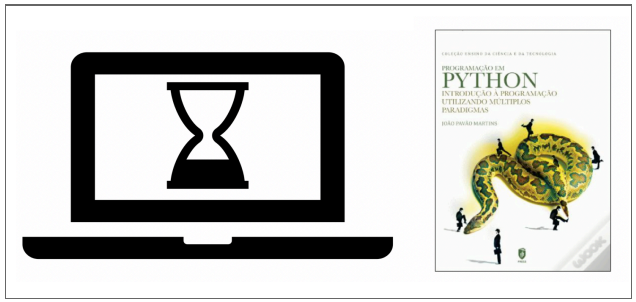
```
CPU times: user 38.7 ms, sys: 1.35 ms, total: 40 ms
Wall time: 39.3 ms
True
```

Listas - Considerações sobre eficiência

- Para compararmos a eficiência de algoritmos temos de analisar a **ordem de crescimento** dos recursos necessários em função do tamanho da entrada, i.e., a sua complexidade computacional no:
    - tempo
    - espaço
- Para caracterizar os tempos de execução dos algoritmos utilizamos uma notação assimptótica chamada **Omaiúsculo** que permite estabelecer taxas de crescimento em função do tamanho da entrada, ex:
    - Procura linear $O(n)$; Procura binária $O(log(n))$; Bubble sort $O(n^2)$
- Para esta análise é importante conhecer a **complexidade das operações sobre várias entidades computacionais em Python**, nomeadamente sobre listas.

Listas - Tarefas próximas aulas

- Trabalhar matéria apresentada hoje:
    - Experimentar todos os programas dos slides
- Ler capítulo 8 do livro da UC: Dicionários
- Nas aula de problemas L08 ==> listas




```
In [ ]:
```