Fundamentos da Programação

Algoritmos de Procura e de Ordenação

Aula 12

José Monteiro

(slides adaptados do Prof. Alberto Abad)

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 [3]:

```
def linearsearch(l, x):
    for i in range(len(l)):
        if l[i] == x:
            return i
    return -1


%timeit -n 1000 linearsearch([1,2,3,7], 7)
%timeit -n 1000 (7 in [1,2,3,7])
```

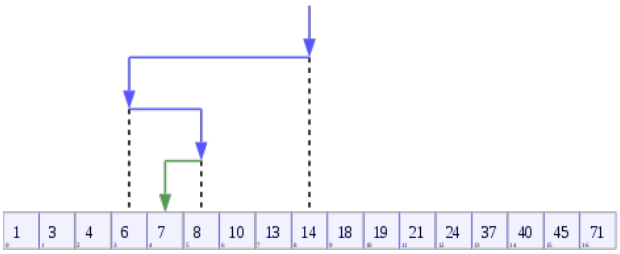
```
627 ns ± 137 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
66.8 ns ± 1.46 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

- 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?

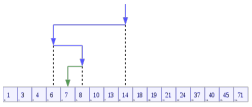
Algoritmos de Procura - Procura Binária

- Podemos fazer melhor se a lista estiver ordenada!!



Algoritmos de Procura - Procura Binária

- Podemos fazer melhor se a lista estiver ordenada!!



In [17]:

```
def binsearch(l, x):
    left = 0
    right = len(l) - 1

    while left <= right:
        mid = left + (right - left)//2
        if x == l[mid]:
            return mid
        elif x > l[mid]:
            left = mid + 1
        else:
            right = mid - 1
    return -1

from random import shuffle
l = list(range(1000))
r = l[:]
#shuffle(r)

%timeit -n 1000 linearsearch(r, 1)
%timeit -n 1000 binsearch(l, 1)
```

```
491 ns ± 25.8 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
2.2 µs ± 122 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

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 [8]:
```
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 [9]:
```
from random import shuffle
nums = list(range(1000))
shuffle(nums)

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

nums1=nums[:]
%time bubblesort(nums1)
print(nums1 == sorted(nums1))
%time linearsearch(nums, 436)
```

```
CPU times: user 90.4 ms, sys: 3.67 ms, total: 94 ms
Wall time: 93.2 ms
True
CPU times: user 55 µs, sys: 0 ns, total: 55 µs
Wall time: 58.2 µs
```

Out[9]:
```
971
```

Algoritmos de Ordenação - _Shell Sort_

In [10]:

```
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[:]
%time bubblesort(nums1)
print(nums1 == sorted(nums1))

nums2=nums[:]
%time shellsort(nums2)
print(nums2 == sorted(nums2))
```

```
CPU times: user 87.8 ms, sys: 3.1 ms, total: 90.9 ms
Wall time: 90.5 ms
True
CPU times: user 6.61 ms, sys: 111 µs, total: 6.73 ms
Wall time: 6.8 ms
True
```

Algoritmos de Ordenação - _Selection Sort_

In [11]:

```
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]

nums = list(range(10))
shuffle(nums)

ind = selectionsort(nums)
print(nums, ind)

# print(nums3 == sorted(nums3))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] None
```

Algoritmos de Ordenação - _Insertion Sort_

In [14]:

```
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

nums4=nums[:]
%time insertionsort(nums4)
print(nums4)
```

```
CPU times: user 7 µs, sys: 1 µs, total: 8 µs
Wall time: 9.06 µs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listas - Considerações sobre Eficiência

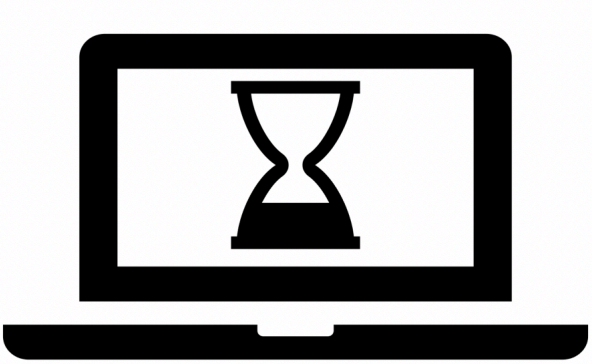
- Para compararmos a eficiência de algoritmos temos de analisar a *ordem de crescimento* dos recursos necessários em função do tamanho da entrada, *i.e.*, a sua complexidade computacional no:
  - tempo
  - espaço

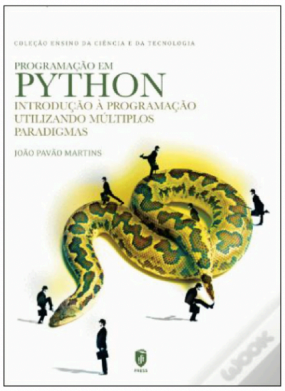
- Para caracterizar os tempos de execução dos algoritmos utilizamos uma notação assimptótica chamada *Omaiúsculo* que permite estabelecer taxas de crescimento em função do tamanho da entrada, ex:
  - Procura linear $O(n)$; Procura binária $O(log(n))$; Bubble sort $O(n^2)$

- Para esta análise é importante conhecer a complexidade das operações sobre várias entidades computacionais em Python, nomeadamente sobre listas.

Listas - Tarefas próxima semana

- Trabalhar matéria apresentada hoje:
  - Experimentar todos os programas dos slides
- Ler Capítulo 8 do livro da UC: Dicionários
- Projeto!!
- Nas aulas de problemas ==> listas





In [ ]: