



Fundamentos de Programação

António J. R. Neves

João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática

Universidade de Aveiro



Summary

- Searching
 - Sequential search
 - Binary search
- Sorting
- Functions as arguments
- Lambda expressions



Searching

- Searching for an element X in a list L (or some other sequence) is a common operation in many problems.
 - Sometimes we just need to check if the element is there.(*)
In Python, we can do this with: `X in L`
 - Other times we need to know where it is.
In Python, we can do this with: `L.index(X)`.
 - These operations are simple, but they can be **expensive**: it takes time (and energy) to search a very large list!
- (*) Note that if all we need is checking membership, then using a set or a dictionary is much faster than a list!



Sequential search

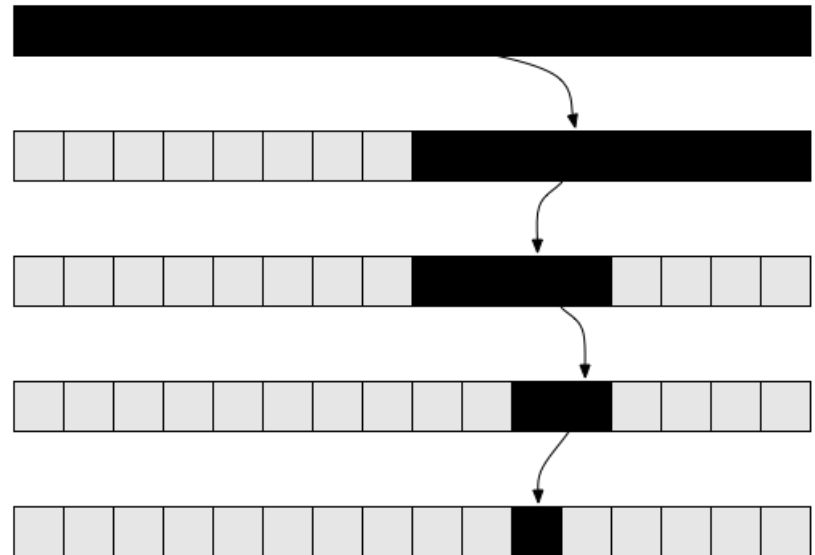
- A **sequential search** scans a sequence from start to end (or from the end to the start).

```
def seqSearch(lst, x):  
    """Return k such that x == lst[k], or None if no such k."""  
    for i in range(len(lst)):  
        if x == lst[i]:  
            return i  
    return None
```

- This is what the `index` method and the `in` operator do.
- Finding an element in a list of length N requires up to N comparisons.

Binary search

- If the sequence is sorted, $L[0] \leq L[1] \leq \dots \leq L[-1]$, there's a much better way to search!
 1. Compare X to the middle element of L .
 2. If X is smaller, search only in the first half of L .
 3. If X is larger, search only in the second half.
- This is the **binary search** algorithm.
- This is much better:
 - $N=15 \Rightarrow$ just 4 comparisons.
 - $N=31 \Rightarrow$ 5 comparisons
 - $N=1023 \Rightarrow$ 10 comparisons.
 - $N \sim 1 \text{ million} \Rightarrow$ 20 comparisons!
- If $N < 2^k \Rightarrow k$ comparisons.





Binary search (2)

- Binary search for exact match (stops when equal).

```
def binSearchExact(lst, x):  
    """Find k such that x == lst[k]. (Or None if no such k.)"""  
    first = 0                # first index that could be solution  
    last = len(lst)         # first index that cannot be solution  
    while first < last:  
        mid = (first+last)//2  
        if x < lst[mid]:  
            last = mid  
        elif x > lst[mid]:  
            first = mid+1  
        else:  
            return mid  
    return None
```

- This works exactly like `seqSearch`, but is much faster!
- With a minor modification, we can make it slightly faster.



Binary search (3)

- Binary search. (Equivalent to `bisect.bisect_left`.)

```
def binSearch(lst, x):  
    """Find k such that: lst[k-1] < x <= lst[k] (not quite!)."""  
    first = 0                # first index that can be result  
    last = len(lst)         # last index that can be result  
    while first < last:  
        mid = (first+last)//2  
        if x <= lst[mid]:    # (just 1 comparison inside loop!)  
            last = mid  
        else:  
            first = mid+1  
    return first
```

- If x is not found, still returns index k of where x should be!
- If $k < \text{len}(\text{lst})$ and $x == \text{lst}[k]$, then we know x was found.
- This is slightly faster, in general.



Sorting

- A sorted sequence is much faster to search.
- Sorting is putting the elements of a sequence in order.
- In Python, use the `sorted` function or the list `sort` method.

```
L.sort()           # Modifies list L in-place  
L2 = sorted(L)     # Creates L2. L is not modified!
```

- `sorted` returns a list, but takes any kind of collection.

```
sorted('banana')   #-> ['a', 'a', 'a', 'b', 'n', 'n']  
N = [9, 7, 2, 8, 5, 3]  
print(sorted(N))   #-> [2, 3, 5, 7, 8, 9]  
L = ["maria", "carla", "anabela", "antonio", "nuno"]  
print(sorted(L))  
#-> ['anabela', 'antonio', 'carla', 'maria', 'nuno']
```




Sorting criteria

- These functions can sort by different **criteria**.

```
L = ["Mario", "Carla", "anabela", "Maria", "nuno"]
print(sorted(L))    # lexicographic sort
#-> ['Carla', 'Maria', 'Mario', 'anabela', 'nuno']
print(sorted(L, key=len))    # sort by length
#-> ['nuno', 'Mario', 'Carla', 'Maria', 'anabela']
print(sorted(L, key=str.lower))    # case-insensitive
#-> ['anabela', 'Carla', 'Maria', 'Mario', 'nuno']
```

- The optional `key` argument receives a function to sort the elements by.
- The key function is applied to each element and the results are compared to establish the order.
- To reverse the order, use the `reverse=True` argument.



Sorting complex data

- Lists of tuples can be sorted, too.

```
dates = [(1910, 10, 5, 'Republic'),  
         (1974, 4, 25, 'Liberty'),  
         (1640, 12, 1, 'Independence')]  
print(sorted(dates))    # "lexicographic" order
```

- Tuples are compared like strings: left-to-right.
- For a different order, use the `key` argument.

```
sorted(dates, key=lambda t: t[3])    #by name  
sorted(dates, key=lambda t: (t[1],t[2])) #by month,day
```

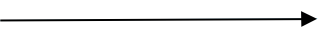
- We're using lambda expressions here!



Lambda expressions

- **Lambda expressions** define simple anonymous functions.

```
sq = lambda x: x**2  
sq(5)    #-> 25  
add = lambda x,y: x+y
```



```
# Same as:  
def sq(x):  
    return x**2
```

- The result must be an expression. No statements allowed!
- Should only be used for simple functions.
- They're useful to pass as arguments (such as key=...).
- Exercise: use a lambda expression to sort names by length, then alphabetically.

```
sorted(L, key=lambda s: (len(s),s))  
    #-> ['nuno', 'Carla', 'Maria', 'Mario', 'anabela']
```

Insertion sort

- There are lots of sorting algorithms. One of the simplest is called insertion sort.
- The insertion sort algorithm:
 1. Assume the first K elements are sorted. $L[K]$ is not.
 2. Save $L[K]$ in T .
 3. Move every $L[J] > T$ to $L[J+1]$, starting from $J=K-1$ down.
 4. Put T into the vacant slot.
 5. Now, increment K and repeat.

