Fundamentos de Programação

António J. R. Neves
João Rodrigues
Osvaldo R. Pacheco

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 <u>if</u> the element is there.(*)
    In Python, we can do this with: `X in L`
  - Other times we need to know <u>where</u> it is.
    In Python, we can do this with: `L.index(X)`.
- These operations are simple, but they can be **slow**: it takes time (and energy) to search a very large list!

(*) Note that if all we need is to check 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).

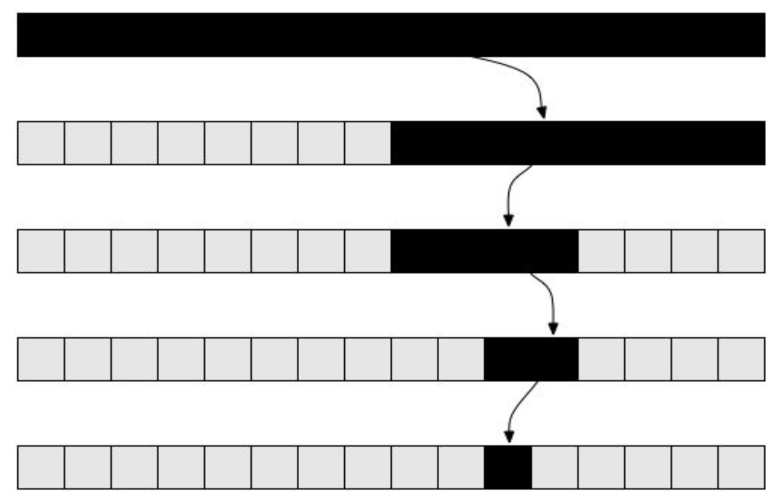
```
def seqSearch(lst, x):
    """Return first k such that x == lst[k]. (Or None if no such k.)"""
    for k in range(len(lst)):
        if x == lst[k]:
            return k
    return None
```

Play ▸

- That is how the `index` method and the **`in`** operator work.
- Finding an element in a list of length N requires up to N comparisons.

Binary search

- If the sequence is sorted, `L[0]<= L[1] <= ... <= L[-1]`, then there's a much better way to search!
  1. Compare X to the element in the middle 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.
- It is much better than sequential:
  - N=15 => just 4 comparisons.
  - N=31 => 5 comparisons
  - N=1023 => 10 comparisons.
  - N ~ 1 million => 20 comparisons!
- If $N < 2^k$ => k comparisons.



Binary search: implementation 1

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

```
# The lst must be sorted for this to work!
def binSearchExact(lst, x):
    """Find k such that x == lst[k]. (Or None if no such k.)"""
    first = 0           # first index that could be result
    last = len(lst)     # first index that cannot be result
    while first < last:
        mid = (first + last) // 2
        elem = lst[mid]
        if x > elem:
            first = mid+1
        elif x < elem:
            last = mid
        else:
            return mid
    return None
```

Play ▸

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

Binary search: implementation 2

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

```
def binSearch(lst, x):
    first = 0               # first index that can be result
    last = len(lst)         # last index that can be result
    while first < last:
        mid = (first + last) // 2
        elem = lst[mid]
        if x > elem:        # (just 1 comparison inside loop!)
            first = mid + 1
        else:
            last = mid
    return first
```

Play ▸

- If x is not found, still returns <u>index where x should be</u>!
- If k < len(lst) and x == lst[k], then we know x was found.
- This is slightly faster, in general.

Using the `bisect` functions

- The bisect module includes functions that perform binary search in a sorted list.

```python
import bisect

lst = [10, 20, 20, 30, 40, 50, 60, 70, 80, 90]

# Using bisect to search values
I40 = bisect.bisect_left(lst, 40)
print(lst[I40] == 40)

I65 = bisect.bisect_left(lst, 65)
print(lst[I65] == 65)

I05 = bisect.bisect_left(lst, 5)
I91 = bisect.bisect_left(lst, 91)

# Difference between _left and _right
L20 = bisect.bisect_left(lst, 20)
R20 = bisect.bisect_right(lst, 20)
```

Play ▸

- There are functions to insert in order, too: `bisect.insort`.

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]
S = {"maria", "carla", "anabela", "antonio", "nuno"}
print(sorted(S))
    #-> ['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 <u>function</u> 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
```

- Remember: 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 <u>anonymous functions</u>.

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

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

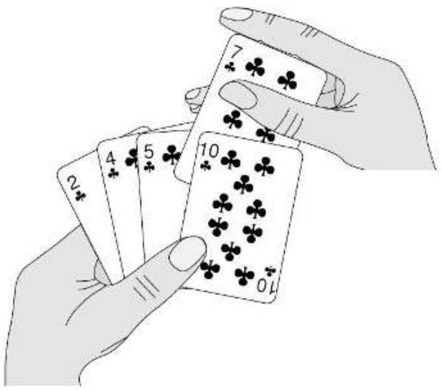
- The result must be an expression. No statements allowed!
- Should only be used for simple functions.
- They're useful to pass as arguments (such as `key=...`).
- Example: use a lambda expression to sort names by length, then alphabetically.

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

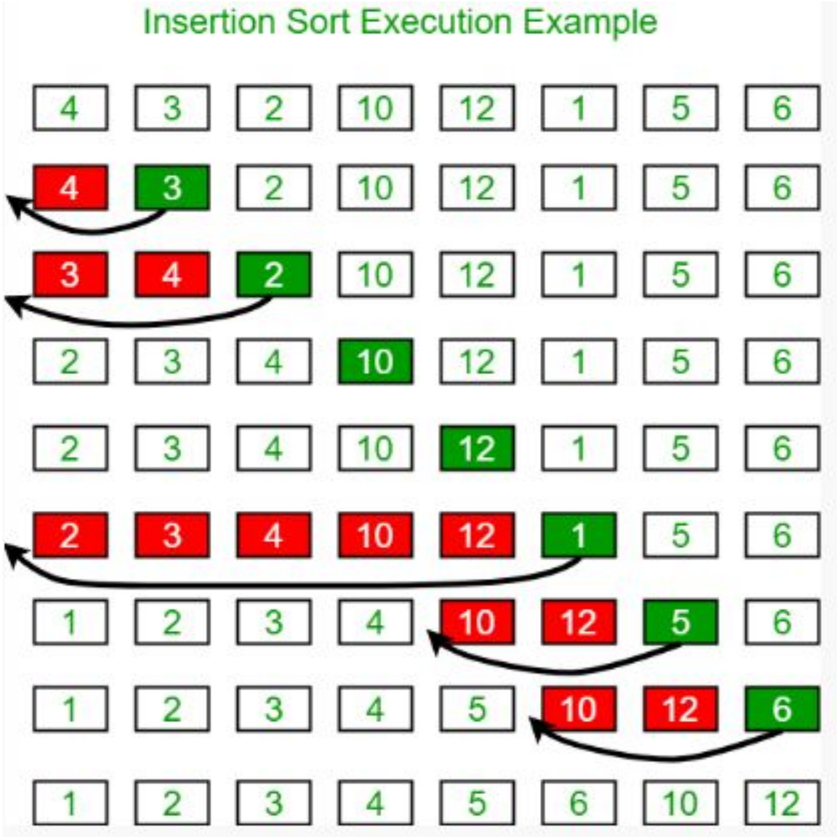
Insertion sort

- 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. (Initially, K=1.)
2. Save L[K] in T.
3. Insert T, in order, into first K elements (overwriting L[K]):
    3.1. Move every L[J]>T to L[J+1], starting from J=K-1 down.
    3.2. Put T into the vacant slot.
4. Now, increment K and repeat.



Insertion sort

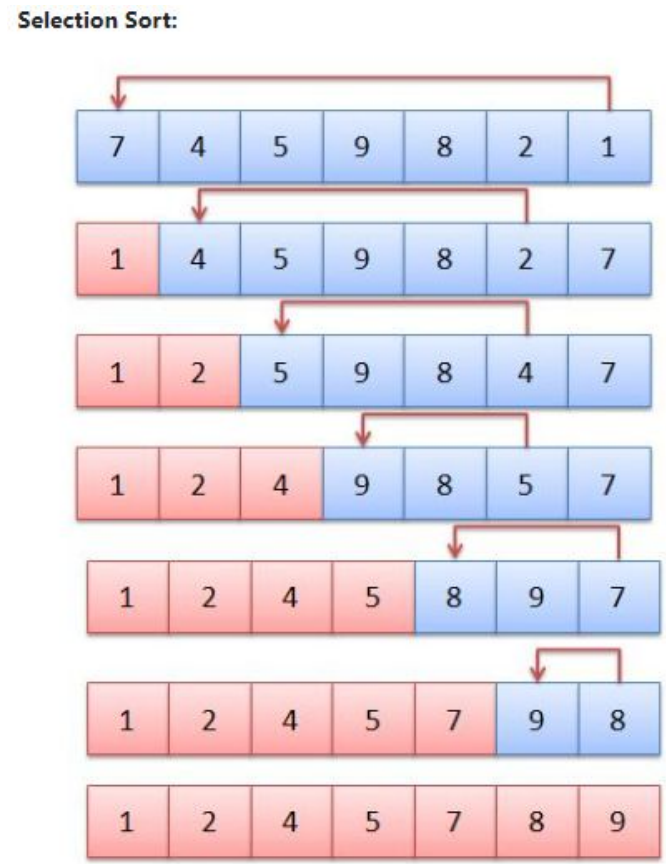


Insertion sort

```
def insertionSort(lst):
    # Traverse elements starting at position 1
    for i in range(1, len(lst)):
        # We know that lst[:i] is sorted
        x = lst[i]    # x is the element to insert next
        # Elements in lst[:i] that are > x must move one position ahead
        j = i - 1
        while j >= 0 and lst[j] > x:
            lst[j + 1] = lst[j]
            j -= 1
        # Then put x in the last emptied slot
        lst[j + 1] = x
        # Now we know that lst[:i+1] is sorted
    return
```

Selection sort

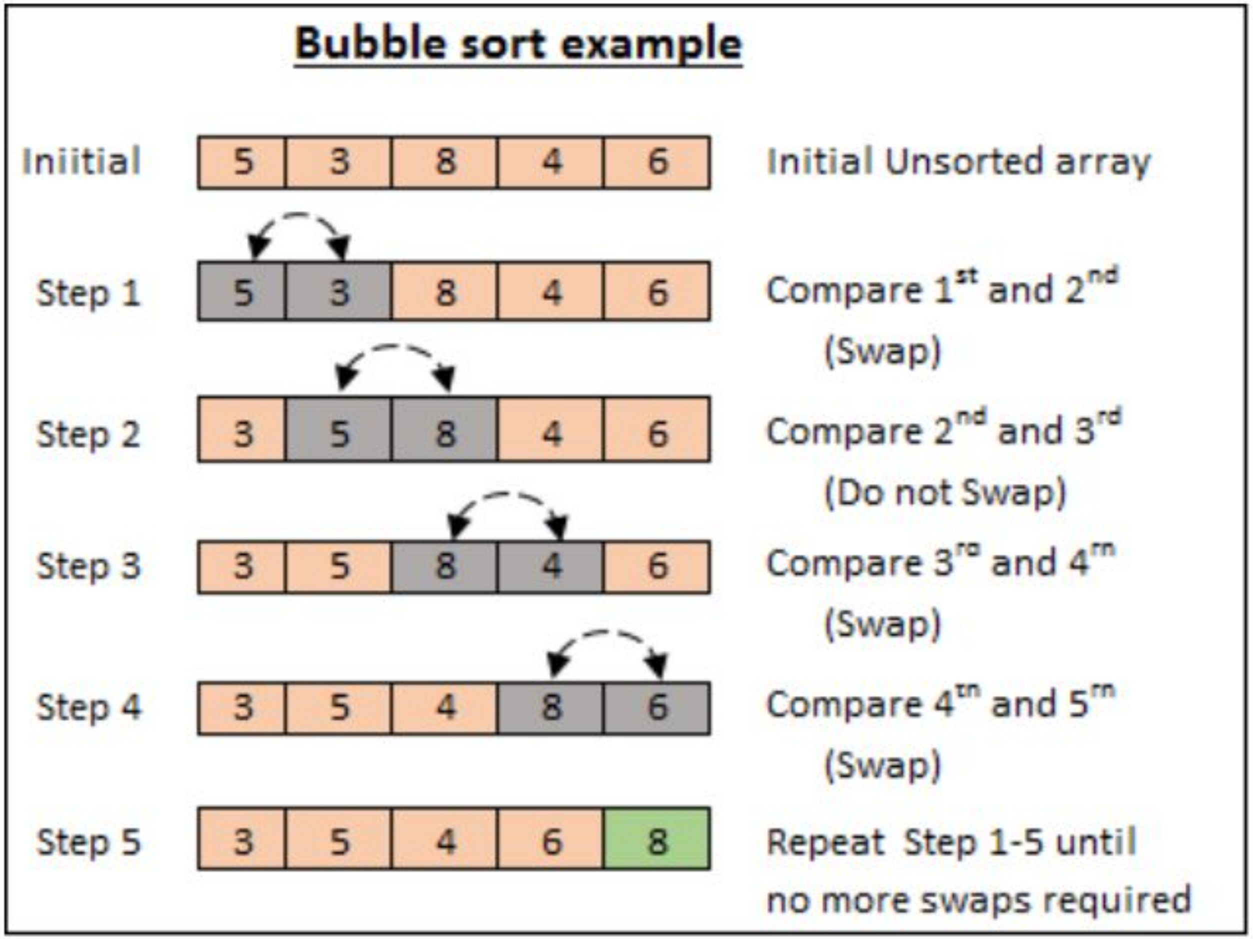
- Selection sort algorithm:
  1. Find position k of the smallest value in L[i:]. (Initially, i = 0).
  2. Swap L[i] with L[k].
  3. Now, increment i and repeat.



Selection sort

```
def selectionSort(lst):
    for i in range(len(lst) - 1):
        # Find position of minimum in lst[i:]
        minpos = i  # first element is the minimum so far
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[minpos]:
                minpos = j  # found new minimum
        # Swap [i] with [minpos] (if not the same)
        if i != minpos:
            aux = lst[i]
            lst[i] = lst[minpos]
            lst[minpos] = aux
    return
```

Bubble sort


Bubble sort example
Iniitial
5
3
8
4
6
Initial Unsorted array
Step 1
5
3
8
4
6
Compare 1st and 2nd
(Swap)
Step 2
3
5
8
4
6
Compare 2nd and 3rd
(Do not Swap)
Step 3
3
5
8
4
6
Compare 3rd and 4th
(Swap)
Step 4
3
5
4
8
6
Compare 4th and 5th
(Swap)
Step 5
3
5
4
6
8
Repeat Step 1-5 until
no more swaps required

Bubble sort

```
def bubbleSort(lst):
    end = len(lst) - 1  # end of sublist that must be sorted
    # lst[:end] may not be sorted yet...
    while end > 0:
        lastswap = 0
        for j in range(end):
            if lst[j] > lst[j + 1]:
                aux = lst[j]
                lst[j] = lst[j + 1]
                lst[j + 1] = aux
                lastswap = j    # remember where we swapped last
        # Now we know lst[lastswap:] must be sorted
        end = lastswap  # next pass must only go that far
    return
```