Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #9 - Searching, sorting
(plus lambda expressions)

Overview

- The need for search and sort

- Searching
  - Sequential search
  - Binary search
- Sorting

- Functions as arguments
- Lambda expressions

Problem

- Scenario:

  You’re organizing a school event and have a long list of registered participants.

  You want to check if a specific person is on the list and have an ordered list

- Task: Write a Python program that:
  - Tells the user whether the name is in the list.
  - Create a list ordered by family name

What do we need?

- A fast way to know if the name is in the list
- A (fast) way to order the list

SEARCH

Searching

- Searching for an element X **in a list** (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 list`
- Other times we need to know where it is located.
  - In Python, we can do this with: `list.index(x)`
- These operations are simple, but they can be slow
  - it takes time (and energy) to search a very large list!
- NOTE: If all we need is to check membership, then using a set or a dictionary is much faster than a list!

Search algorithms

- Search algorithms are techniques used to locate specific data within a collection.

- They vary in complexity, speed, and suitability depending on the data structure and whether the data is sorted.

Search algorithms

- Search algorithms can be broadly categorized into simple, structured, hash-based and graph-based types.

Search Strategy Category	Representative Algorithms	Best For
Unstructured Search	Sequential Search	Small or unsorted datasets
Structured Search	Binary, Jump, Interpolation, Exponential Search	Fast lookup in sorted arrays
Hash-Based Search	Hash Table Lookup	Constant-time access in key-value stores
Graph-Based Search	BFS, DFS, A* Search	Pathfinding, network traversal, puzzle solving

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 ▸

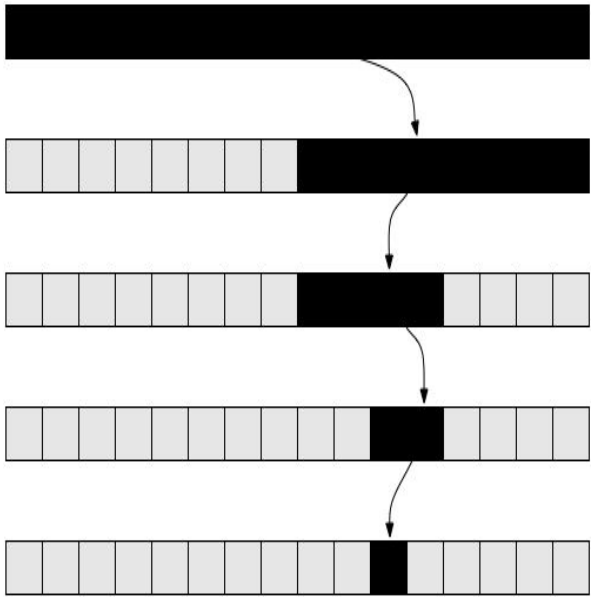
- That is how the `index()` method and the `in` operator work.

- Finding an element in a list of length N requires up to N comparisons.

Binary search

If the sequence is sorted there's a much better way to search!

1. Compare x to the element in the middle of the list.
2. If x is smaller, search only in the first half of the list.
3. If x is larger, search only in the second half.



This is the binary search algorithm.

Binary search

- It is much better than sequential:

N = len(list)	Number of comparisons
15	Just 4 comparisons
31	5
1023	10
~ 1 million	20
…	
$2^{k-1} <= N < 2^k$	k

SEARCH IN PYTHON

Binary search: implementation 1

- 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 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 much faster!
- With a minor modification, we can make it slightly faster.

Binary search: implementation 2

```
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 index where x should be!
- If k < len(lst) and x == lst[k], then we know x was found.
- This is slightly faster, in general.

Using bisect module functions

- The bisect module includes functions that perform binary search in a sorted list.
- `bisect_left(list,x)` locates the insertion point for x in list to maintain sorted order using binary search (2nd implementation)
- `bisect_right(list,x)` is similar to `bisect_left()`, but returns an insertion point after (to the right of) any existing entries of x in list

```
import bisect
lst = [10, 20, 20, 30, 40, 50, 60, 70, 80, 90]
# Using bisect to search values
I40 = bisect.bisect_left(lst, 40)       # I40 = 4
print(lst[I40] == 40)
I65 = bisect.bisect_left(lst, 65)       # I65 = 7
print(lst[I65] == 65)

I05 = bisect.bisect_left(lst, 5)        # I05 = 0
I91 = bisect.bisect_left(lst, 91)       # I05 = 10

# Difference between _left and _right
L20 = bisect.bisect_left(lst, 20)       # L20 = 1
R20 = bisect.bisect_right(lst, 20)      # R20 = 3
```

list

0	1	2	3	4	5	6	7	8	9
10	20	20	30	40	50	60	70	80	90

Play ▶

- There are also functions to insert in order: bisect.insort().

SORTING

Sorting?

- Sorting is **putting the elements** of a sequence **in order**.

- Why does it matter?
  - Organizes data for efficient searching and processing
    - A sorted sequence is much faster to search.
  - Essential for databases, spreadsheets, and search engines
  - Foundation for more advanced algorithms

Sorting algorithms

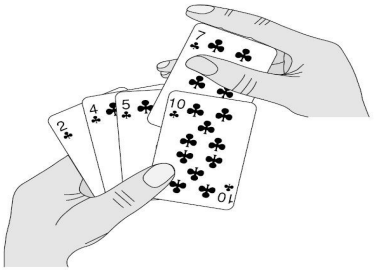
- There are many ways to sort data…
- From simple methods that are easy to understand …
- To fast algorithms that power modern software.

Types of Sorting Algorithms

- Simple Algorithms
  - Best for small datasets and teaching fundamentals
    - **Insertion Sort**: Builds the sorted list one item at a time
    - **Selection Sort**: Selects the smallest (or largest) element and places it in order
    - **Bubble Sort**: Repeatedly swaps adjacent elements if they're in the wrong order
- Efficient Algorithms:
  - Fast and widely used in real-world applications
    - Merge Sort: Divides the list, sorts each half, and merges them
    - Quick Sort: Uses a pivot to partition and recursively sort
- Specialized Algorithms:
  - Optimized for specific data types or constraint
    - Heap Sort: Uses a heap data structure to sort efficiently
    - Radix Sort: Sorts numbers digit by digit

Insertion sort

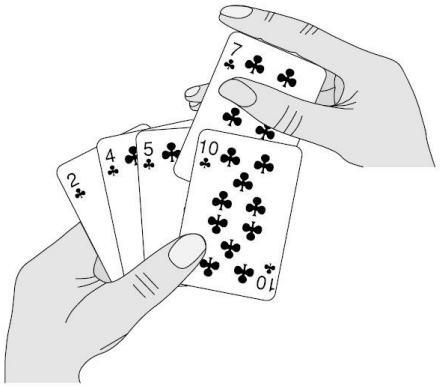
- One of the simplest is called insertion sort.
- Is commonly used to sort a hand of cards



6 5 3 1 8 7 2 4

Insertion sort execution example

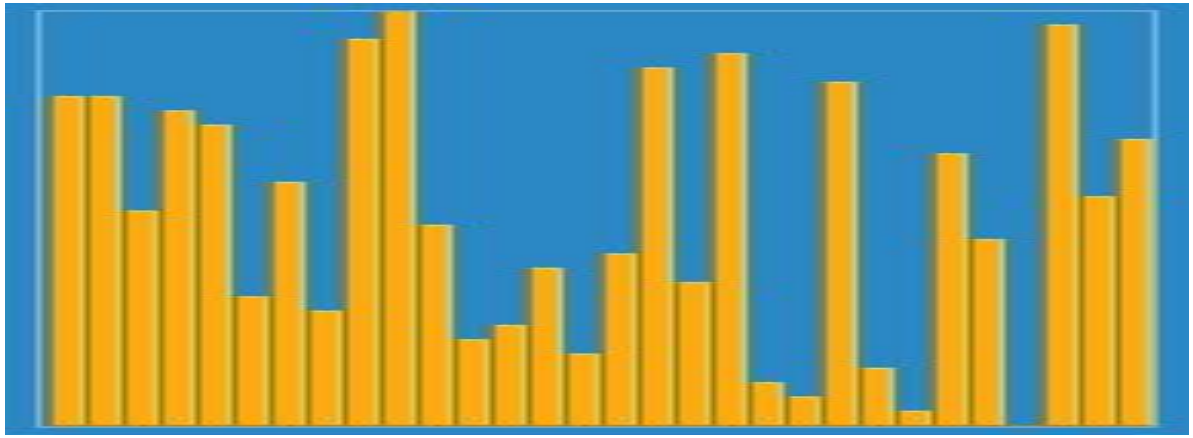
The insertion sort algorithm



1. Start from the second item in the list
   (the first is considered sorted).

2. Compare it with the items before it.
3. Shift each larger item to the right to make space.
4. Insert the current item into its correct position.

5. Repeat for all following items until the last.



Insertion sort in python

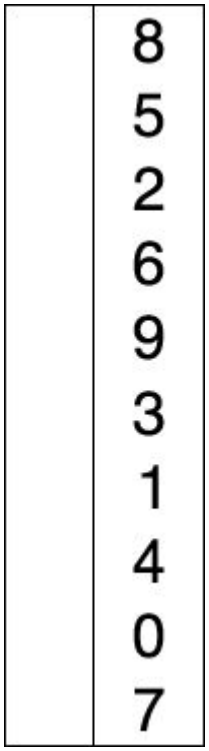
```
def insertionSort(lst):
    # Traverse elements starting at position 1
    for i in range(1, len(lst)):
        # We know that lst[:i] is sorted
        x = lst[i]        # x is the element to insert next
        # Elements in lst[:i] that are > x must move one position ahead
        j = i - 1
        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 lst

example = [7, 9, 3, 2, 10, 1]
print(insertionSort(example))
```

Play ▸

Selection sort

- Divides the input list into two parts:
  - a sorted sublist of items which is built up
  - and a sublist of the remaining unsorted items that occupy the rest of the list.
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list

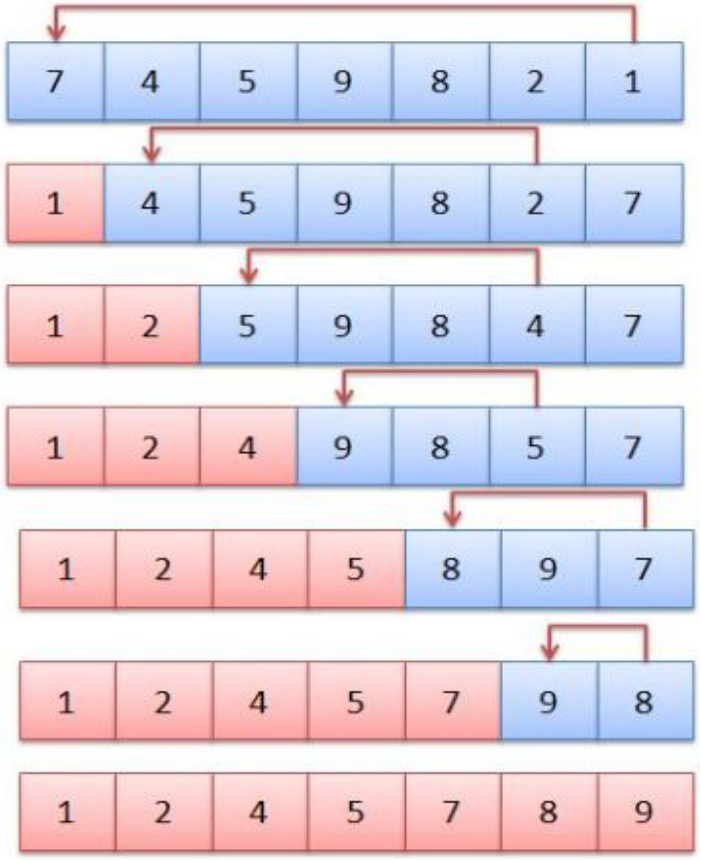
- The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist
  - exchanging (swapping) it with the first unsorted element (putting it in sorted order)



Red is current min.
Yellow is sorted list.
Blue is current item.

Selection sort algorithm

- Find position k of the smallest value in list[i:].
  (Initially, i = 0).
- Swap list[i] with list[k].
- Increment i and repeat.



Selection sort in python

```
def selectionSort(lst):
    for i in range(len(lst) - 1):
        # Find position of minimum in lst[i:]
        minpos = i  # first 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 lst
```

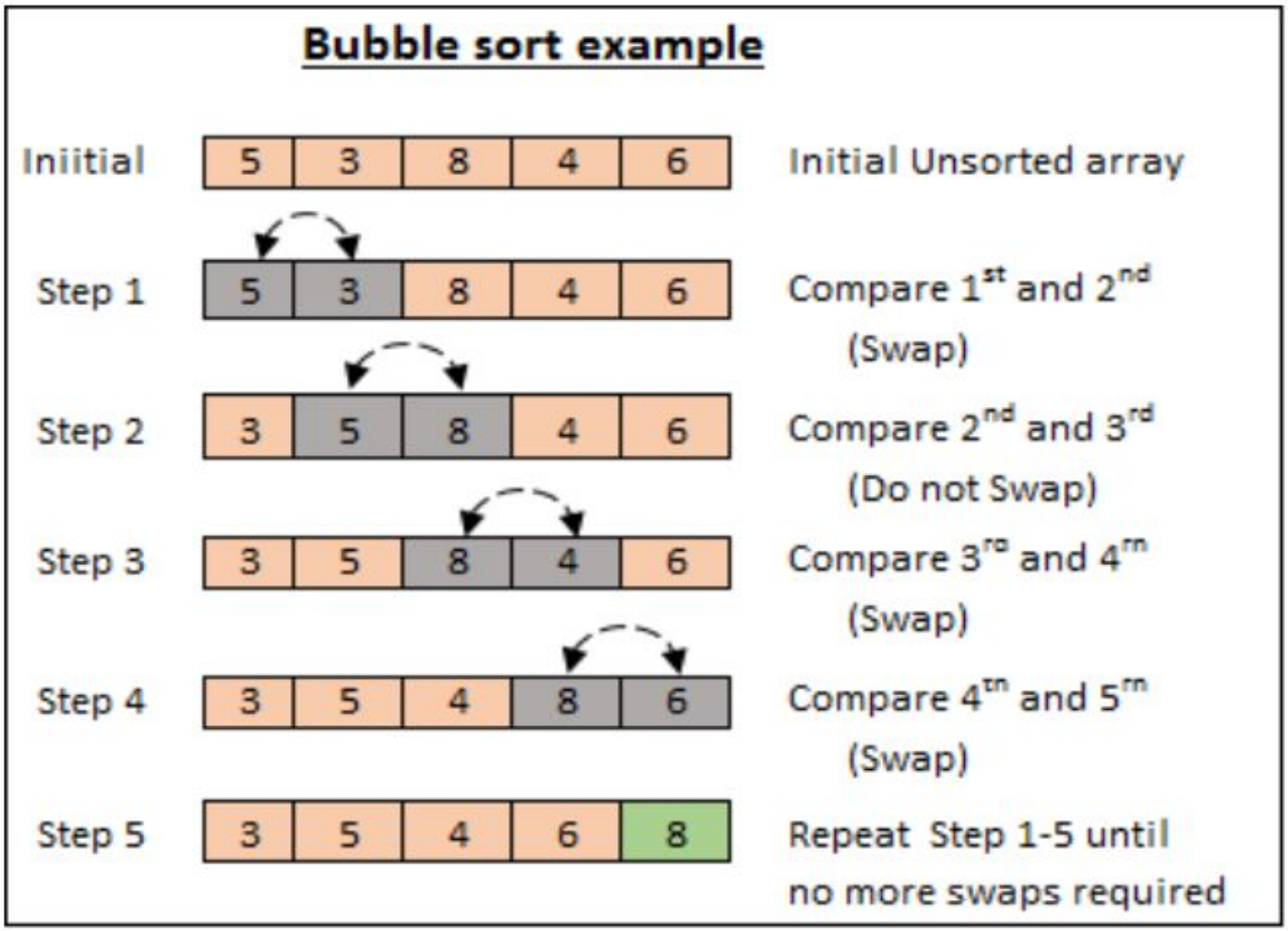
Play ▸

Bubble sort

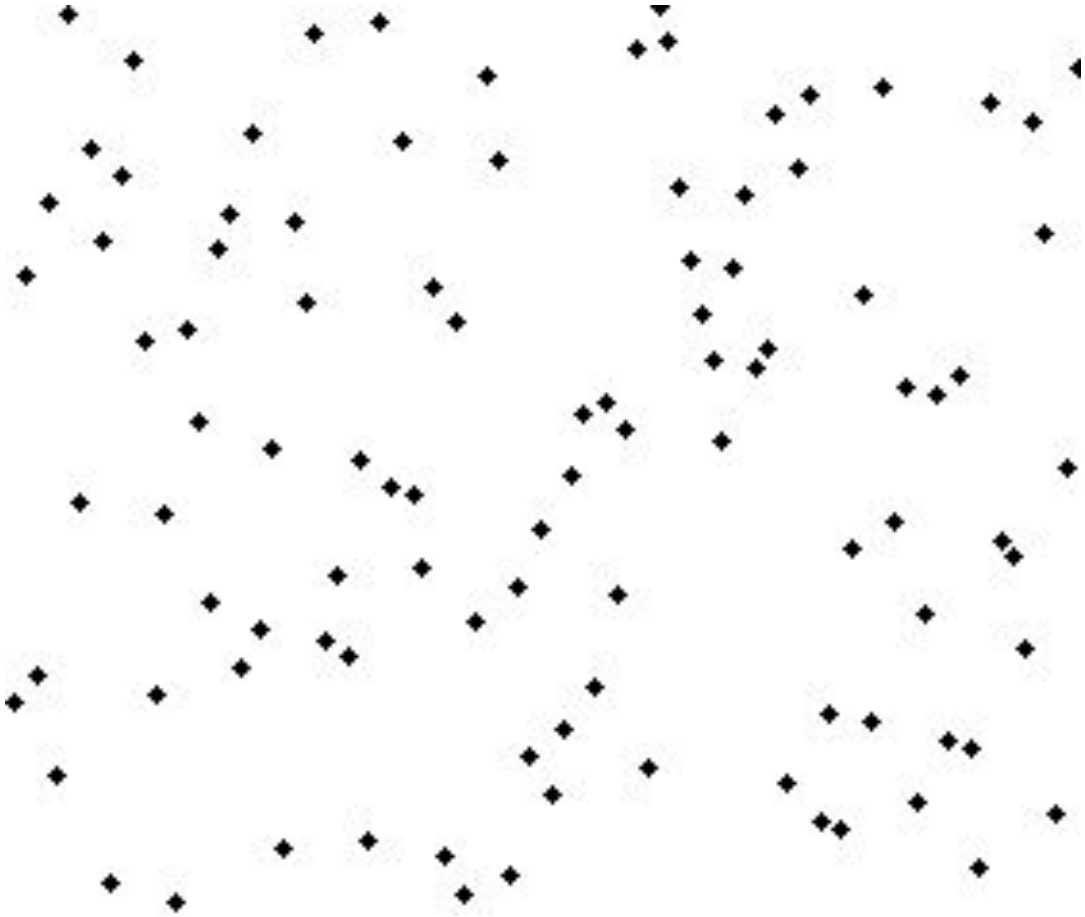
- Repeatedly steps through the input list element by element …
- comparing the current element with the one after it,
- swapping their values if needed

6 5 3 1 8 7 2 4

Bubble sort example



Bubble sort algorithm in action



Bubble sort in python

```
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 lst
```

Play ▸

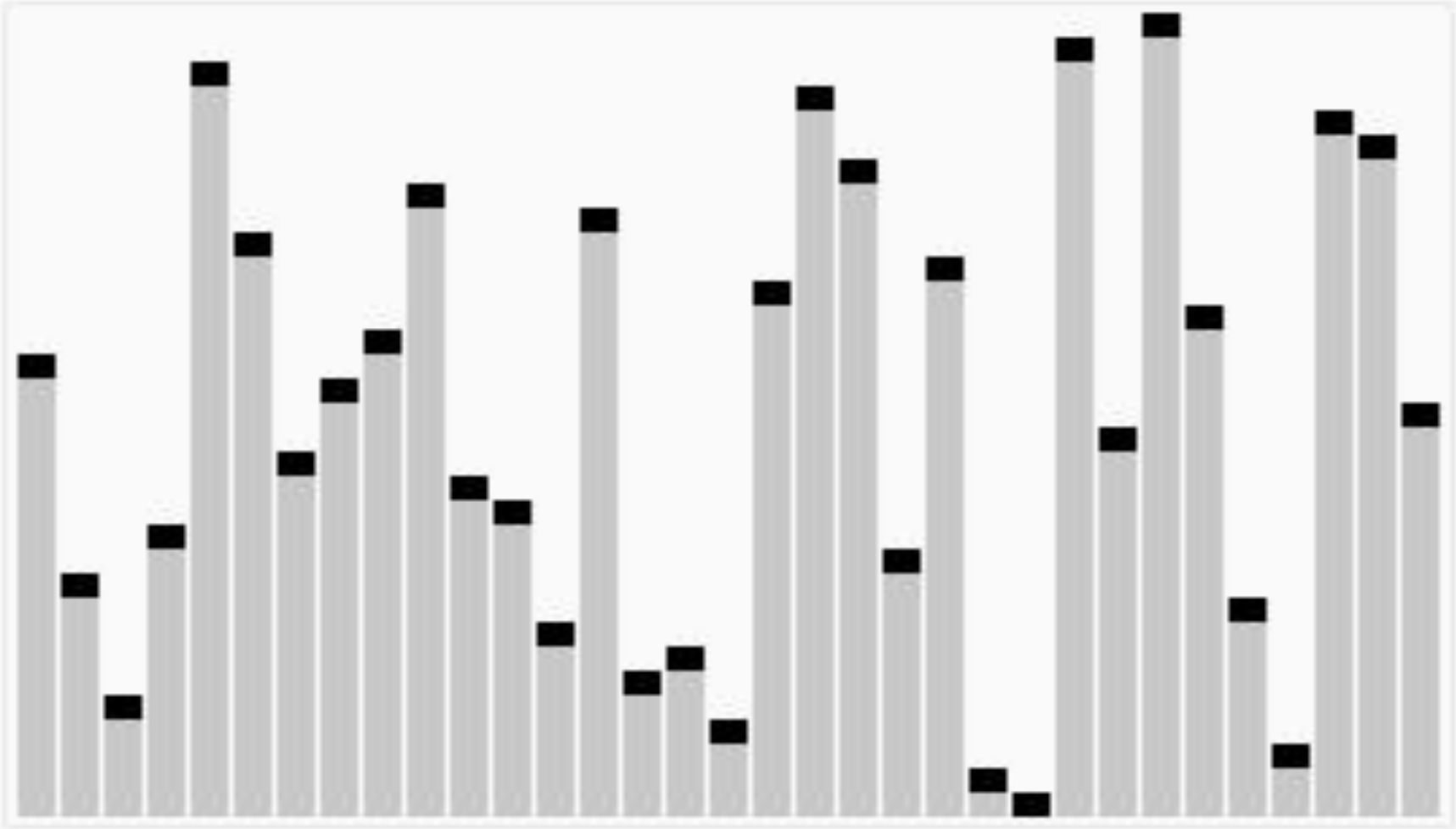
Quicksort

- Quicksort is a fast, efficient sorting algorithm that uses a **divide-and-conquer strategy**
- Invented by C.A.R. Hoare in 1960

- Algorithm:
  - Pick a pivot element from the list.
  - Partition the list into two parts:
    - Elements less than the pivot
    - Elements greater than or equal to the pivot
  - Recursively apply the same process to each part.
  - Combine the sorted parts to get the final sorted list.

- It's known for its speed and is often used in real-world applications.

Quicksort in action



quicksort in python

```
def quicksort(lst):
    if len(lst) <= 1:
        return lst
    # 1 - Pick a pivot element from the list
    pivot = lst[0]

    # 2 - Partition the list into two parts:
    #   Elements less than the pivot
    less = [x for x in lst[1:] if x < pivot]
    #   Elements greater than or equal to the pivot
    greater = [x for x in lst[1:] if x >= pivot]

    # 3 - Recursively apply the same process to each part
    # 4 - Combine
    return quicksort(less) + [pivot] + quicksort(greater)

example = [7, 9, 3, 2, 10, 1]
print(quicksort(example))
```

Play ▸

SORTING IN PYTHON

Sorting lists – `sort()`

- Python lists have a method for sorting: **`sort()`**
- It modifies the list in-place
- Example:

```
# Random list of 10 integers between 1 and 100
numbers = random.sample(range(1, 101), 10)
print("Original list:", numbers)

# Sort the list in ascending order
numbers.sort()

print("Sorted list:", numbers)
```

```
Original list: [88, 62, 68, 49, 14, 23, 18, 13, 81, 31]
Sorted list: [13, 14, 18, 23, 31, 49, 62, 68, 81, 88]
```

Function `sorted()`

- There is also the `sorted()` function.

```
list2 = sorted(list)  # Creates list2.
                      # list is not modified!
```

- `sorted()` takes any kind of collection as input, but always returns a list.

```
sorted('banana') #-> ['a', 'a', 'a', 'b', 'n', 'n’]

numbers = (9, 7, 2, 8, 5, 3)
print(sorted(numbers)) #-> [2, 3, 5, 7, 8, 9]

set_names = {"maria", "carla", "anabela", "antonio", "nuno"}
print(sorted(set_names))
    #-> ['anabela', 'antonio', 'carla', 'maria', 'nuno']
```

Sorting criteria

- Both sort() and sorted() 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.
  - But the elements are swapped, not the results!

- 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
```

- Output:

  [(1640, 12, 1, 'Independence'), (1910, 10, 5, 'Republic'), (1974, 4, 25, 'Liberty’)]

- Remember: tuples are compared like strings: left-to-right.

LAMBDA EXPRESSSIONS

Lambda expressions

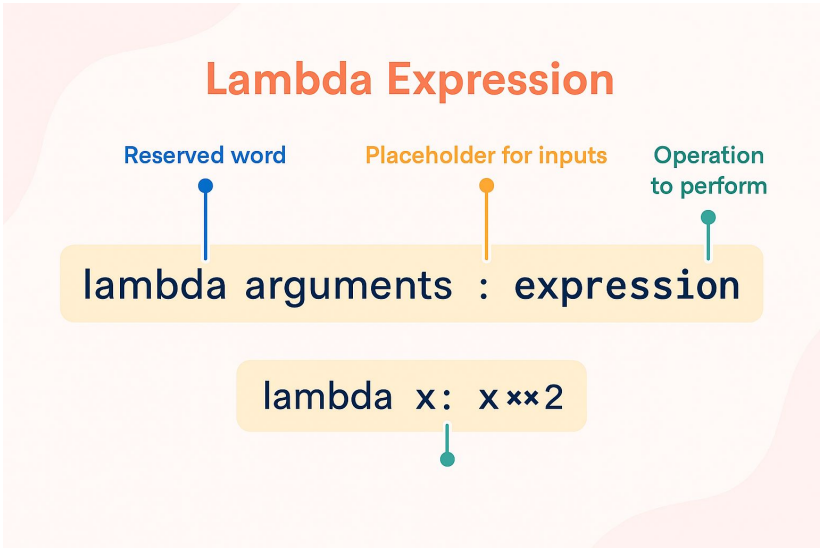
- **Lambda expressions** are anonymous functions
  - functions defined without a name.
- They are used for short, inline operations, often passed as arguments to higher-order functions.
- Common in languages like Python, JavaScript, Java, and functional programming.
- The term comes from Lambda Calculus
  - A formal system in mathematical logic developed by Alonzo Church in the 1930s.
  - Uses the Greek letter λ (lambda) to denote function abstraction.

Lambda expressions in python

- Examples of 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
```



Lambda expressions in python

- 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: using a lambda expression to sort names by length, then alphabetically.

```
list2 = sorted(list, key=lambda s: (len(s), s))
    #-> ['nuno', 'Carla', 'Maria', 'Mario', 'anabela']
```

Examples

```
square = lambda x: x * x
print(square(5))  # Output: 25

add = lambda a, b: a + b
print(add(3, 4))  # Output: 7

#  Lambda in list comprehension
double = lambda x: x * 2
results = [double(x) for x in range(5)]
print(results)  # Output: [0, 2, 4, 6, 8]
```

Sorting and lambda expressions

```
dates = [(1910, 10, 5, 'Republic'),
         (1974, 4, 25, 'Liberty'),
         (1640, 12, 1, 'Independence’)]
```

- As seen before, sorted(dates) sorts in "lexicographic" order.
- For a different order, we can use the key argument and a lambda expression.
  - To sort by event name:

```
dates2 = sorted(dates, key=lambda t: t[3])
```

  - To sort by month and day

```
dates3 = sorted(dates, key=lambda t:(t[1],t[2]))
```

Play ▸

Nice to know, but not essential

ADDITIONAL INFORMATION

Changing criteria for max() and min()

- The max and min functions can use different criteria, too.

```
list = ["Mario", "Carla", "anabela", "Maria", "nuno"]
print(min(list))       # lexicographic minimum
    #-> 'Carla'
print(max(list, key=len)) # element with maximum length
    #-> 'anabela'
```

- Use the key argument just like in sort.

- Example: find the index of the maximum in a list

```
lst = [4, 5, 8, 3, 5, 6]  # the list
ind = range(len(lst))     # sequence of indices to lst
max(ind, key=lambda i: lst[i])  #-> 2
   # finds the index i such that lst[i] is maximum
```

SOME COMMON MISTAKES

And How to Identify & Fix Them

Misusing Search Pattern

- Mistake: Using list.index() without checking if the item exists
- Example: my_list.index('x') → raises ValueError if 'x' not found
- Fix: Use if 'x' in my_list: before calling .index()

Misusing Search Pattern (2)

- Mistake: Confusing find() with index() in strings

- find() returns -1 if not found

  index() raises an error

Sorting Pitfalls

- Mistake: Using `sort()` on a copy unintentionally

- `sorted_list = list.sort()`

  → returns None

- Fix: Use sorted_list = sorted(list) for a new sorted copy

Sorting Pitfalls (2)

- Mistake: Forgetting key= when sorting by attribute
- Example: sorted(data, lambda x: x.age) → ❌
- Fix: sorted(data, key=lambda x: x.age) ✅

Lamba Expression misuse

- Mistake: Using lambdas for complex logic
- Example:

```
lambda x: x if x > 0 else -x
```
→ hard to read

- Fix: Use def for clarity

Recommendations

- Validate existence before searching with .index()
- Use key= explicitly in sorted(), max(), min()
- Prefer def over lambda for complex logic
- Avoid sorting lists with mixed types

RETURNING TO OUR INITIAL PROBLEM …

Possible Solution in Python