Elementary Sorts

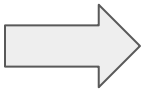
Motivation. Why sorting?

Motivation. Sorting student records

Name	Grade	Phone
Bob	C	314-3125
Alice	B	374-5186
Mark	A	210-8731
Joe	E	244-4640
Kate	C	648-9020
Emily	D	938-9506
Tom	B	546-6046

Motivation. Sorting student records

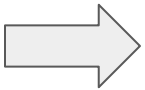
Name	Grade	Phone
Bob	C	314-3125
Alice	B	374-5186
Mark	A	210-8731
Joe	E	244-4640
Kate	C	648-9020
Emily	D	938-9506
Tom	B	546-6046



Name	Grade	Phone
Alice	B	374-5186
Bob	C	314-3125
Emily	D	938-9506
Joe	E	244-4640
Kate	C	648-9020
Mark	A	210-8731
Tom	B	546-6046

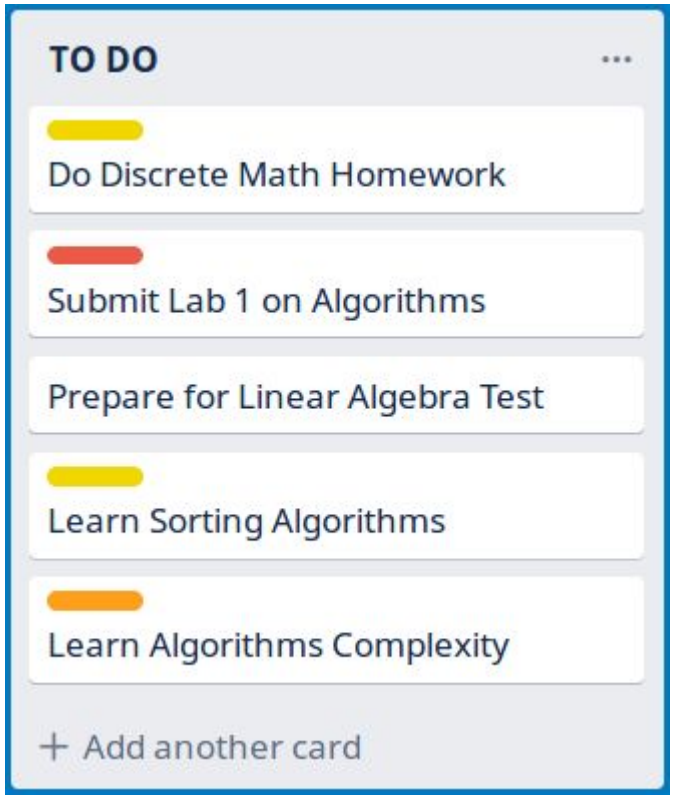
Motivation. Sorting student records

Name	Grade	Phone
Bob	C	314-3125
Alice	B	374-5186
Mark	A	210-8731
Joe	E	244-4640
Kate	C	648-9020
Emily	D	938-9506
Tom	B	546-6046

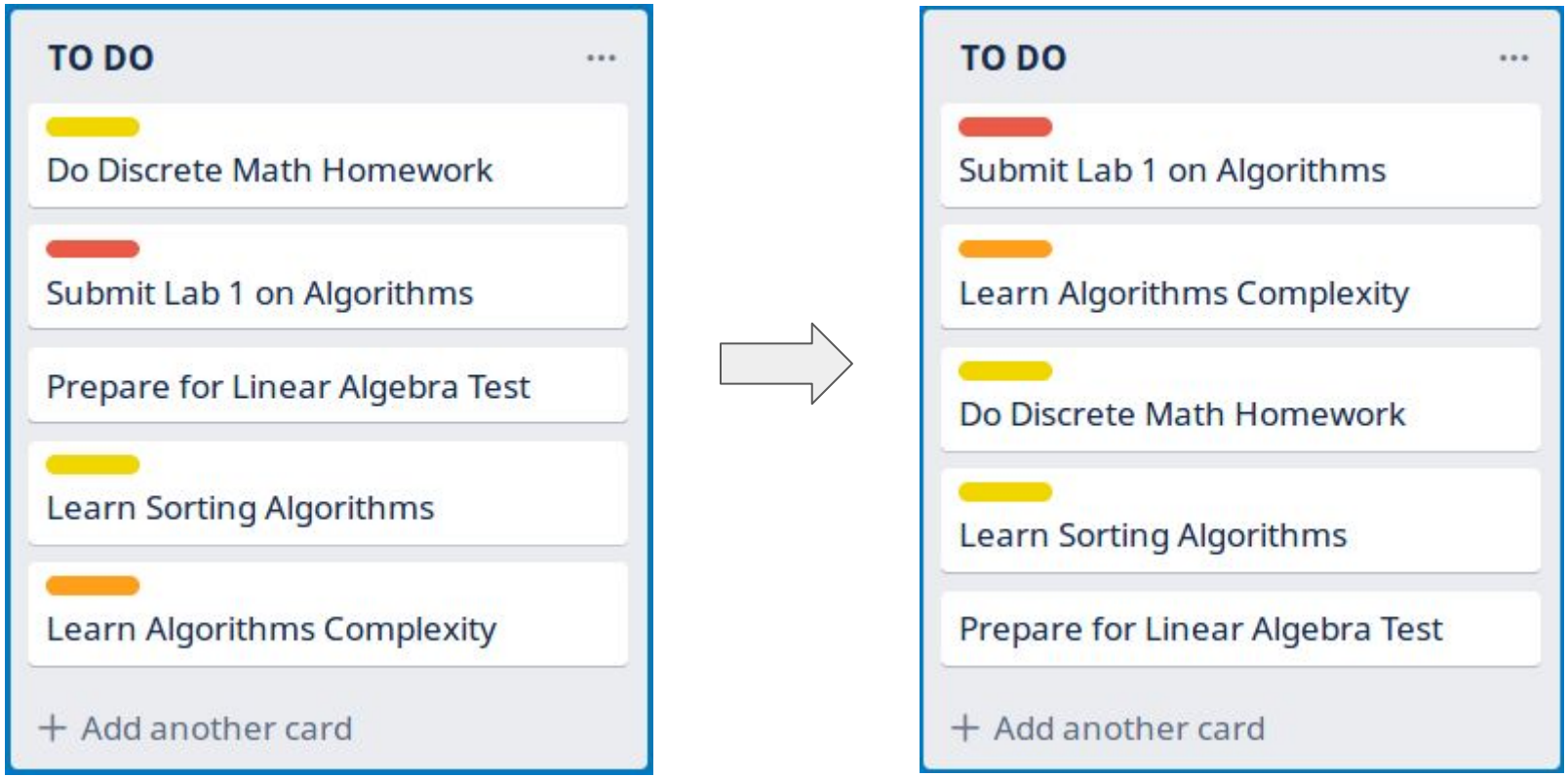


Name	Grade	Phone
Mark	A	210-8731
Alice	B	374-5186
Tom	B	546-6046
Bob	C	314-3125
Kate	C	648-9020
Emily	D	938-9506
Joe	E	244-4640

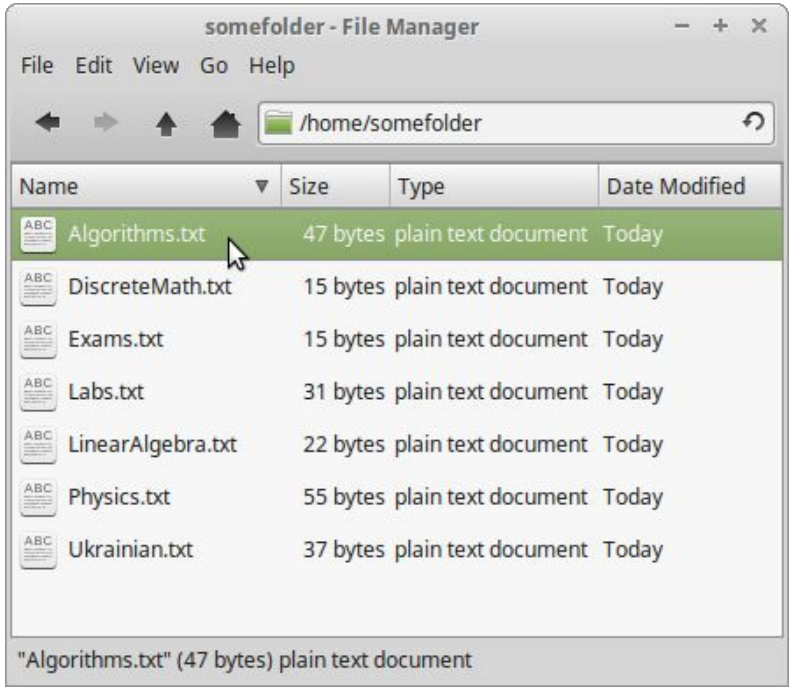
Motivation. Sorting tasks



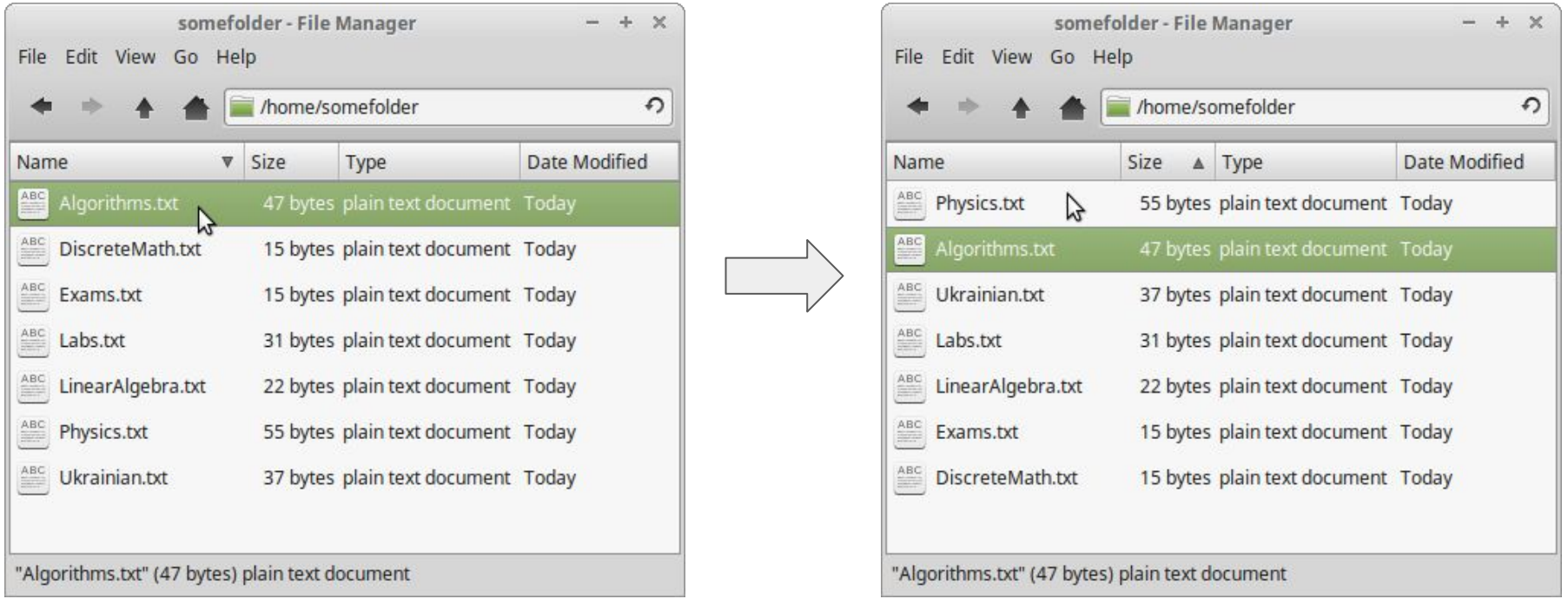
Motivation. Sorting tasks



Motivation. Sorting files



Motivation. Sorting files



When can we sort items?

When can we sort items?

- If we know how to compare them
- Examples
  - Numbers
  - Strings (alphabetically)
  - Dates (chronologically)

Total order

A **total order** is a binary relation ≤ that satisfies:

- Antisymmetry: if $v \leq w$ and $w \leq v$, then $v = w$
- Transitivity: if $v \leq w$ and $w \leq x$, then $v \leq x$
- Totality: either $v \leq w$ or $w \leq v$ or both

Total order

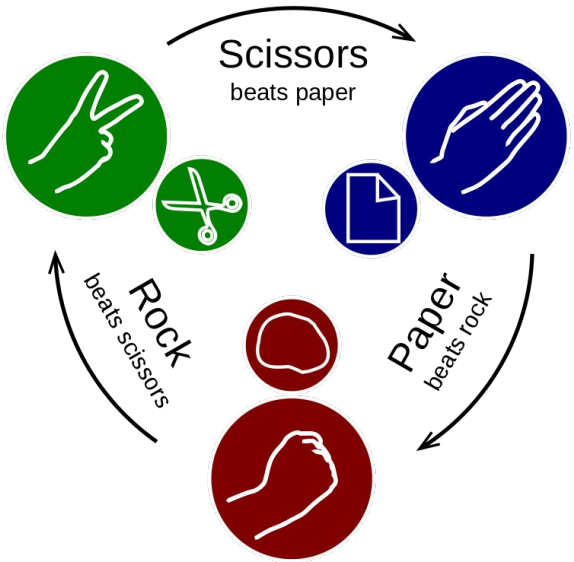
Examples:

- Standard order for natural and real numbers
- Alphabetical order for strings
- Chronological order for dates or times

Total order

Example of absence of total order:

- Rock-Scissors-Paper Game



Okay. But what about Python?

In Python we can use these functions:

- list.sort()
- sorted()

```
numbers = [2,3,1,5,4]

numbers.sort()

print(numbers)
# Outputs: [1, 2, 3, 4, 5]
```

Let’s define class Task

```
class Task:
    def __init__(self, title, priority):
        self.title = title
        self.priority = priority

    def __str__(self):
        return "(Priority: {}, Title: {})".format(self.priority, self.title)
```

And try to sort a list of tasks

```
tasks = [Task("Do Discrete Math Homework", 3),
         Task("Submit Lab 1 on Algorithms", 1),
         Task("Prepare for Linear Algebra Test", 4),
         Task("Learn Sorting Algorithms", 3),
         Task("Learn Algorithms Complexity", 2)]

tasks.sort()

for task in tasks:
    print(task)

# Output:
# (Priority: 3, Title: Do Discrete Math Homework)
# (Priority: 1, Title: Submit Lab 1 on Algorithms)
# (Priority: 4, Title: Prepare for Linear Algebra Test)
# (Priority: 3, Title: Learn Sorting Algorithms)
# (Priority: 2, Title: Learn Algorithms Complexity)
```

What’s the problem?

- Python doesn’t know how to sort such classes properly
- We need to provide additional information:
  - Explicitly pass parameter “key” to sort() method
  - Or overload operation “<”

Extracting comparison key from object using function

```
def extract_priority(task):
    return task.priority

tasks.sort(key=extract_priority)

for task in tasks:
    print(task)

# Output:
# (Priority: 1, Title: Submit Lab 1 on Algorithms)
# (Priority: 2, Title: Learn Algorithms Complexity)
# (Priority: 3, Title: Do Discrete Math Homework)
# (Priority: 3, Title: Learn Sorting Algorithms)
# (Priority: 4, Title: Prepare for Linear Algebra Test)
```

Extracting comparison key from object using lambda

```
tasks.sort(key=lambda task: task.priority)

for task in tasks:
   print(task)

# Output:
# (Priority: 1, Title: Submit Lab 1 on Algorithms)
# (Priority: 2, Title: Learn Algorithms Complexity)
# (Priority: 3, Title: Do Discrete Math Homework)
# (Priority: 3, Title: Learn Sorting Algorithms)
# (Priority: 4, Title: Prepare for Linear Algebra Test)
```

Overloading “<” operation

```
class Task:
    # other methods missed

    def __lt__(self, other): # lt = less than
        return self.priority < other.priority


tasks.sort()

for task in tasks:
    print(task)

# Output:
# (Priority: 1, Title: Submit Lab 1 on Algorithms)
# (Priority: 2, Title: Learn Algorithms Complexity)
# (Priority: 3, Title: Do Discrete Math Homework)
# (Priority: 3, Title: Learn Sorting Algorithms)
# (Priority: 4, Title: Prepare for Linear Algebra Test)
```

Elementary sorts

- Selection sort
- Bubble sort
- Insertion sort

Selection sort. Idea

The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted
2. Remaining subarray which is unsorted

In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray.

Selection sort. Example

```
arr = [64, 25, 12, 22, 11]

# Find the minimum element in arr and place it at beginning
[11, 25, 12, 22, 64]

# Find the minimum element in arr[1:5]
# and place it at beginning of arr[1:5]
[11, 12, 25, 22, 64]

# Find the minimum element in arr[2:5]
# and place it at beginning of arr[2:5]
[11, 12, 22, 25, 64]

# Find the minimum element in arr[3:5]
# and place it at beginning of arr[3:5]
[11, 12, 22, 25, 64]
```

Selection sort. Implementation

```
def selection_sort(arr):
    # Traverse through all array elements
    for i in range(len(arr)):

        # Find the minimum element in remaining unsorted array
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j

        # Swap the found minimum element
        # with the first element of unsorted array
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Selection Sort. Performance

- Worst-case: $O(n^2)$ comparisons, $O(n)$ swaps
- Best-case: $O(n^2)$ comparisons, $O(n)$ swaps
- On average: $O(n^2)$ comparisons, $O(n)$ swaps

Bubble sort. Idea

- Bubble Sort is simple sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order
- The algorithm is named for the way smaller or larger elements "bubble" to the top of the list

Bubble sort. Example

```
arr = [64, 25, 12, 22, 11]

# 64 “bubbles” to the end of arr
[25, 12, 22, 11, 64]

# 25 “bubbles” to the end of arr[0:4]
[12, 22, 11, 25, 64]

# 22 “bubbles” to the end of arr[0:3]
[12, 11, 22, 25, 64]

# 12 “bubbles” to the end of arr[0:2]
[11, 12, 22, 25, 64]
```

Bubble sort. Implementation

```
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n - i - 1):

            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Bubble sort. Performance

- Worst-case: $O(n^2)$ comparisons, $O(n^2)$ swaps
- Best-case: O(n) comparisons, O(1) swaps
  - Need to additionally check whether there were swaps
- On average: $O(n^2)$ comparisons, $O(n^2)$ swaps

Insertion sort. Idea

- Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

Insertion sort. Example

```
arr = [64, 25, 12, 22, 11]

# insert 25 at the correct position of sorted array at beginning
[25, 64, 12, 22, 11]

# insert 12 at the correct position of sorted array at beginning
[12, 25, 64, 22, 11]

# insert 22 at the correct position of sorted array at beginning
[12, 22, 25, 64, 11]

# insert 11 at the correct position of sorted array at beginning
[11, 12, 22, 25, 64]
```

Insertion sort. Implementation

```
def insertion_sort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0:i], that are
        # greater than key, to one position ahead
        # of their current position
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Insertion sort. Performance

- Worst-case: $O(n^2)$ comparisons, $O(n^2)$ swaps
- Best-case: $O(n)$ comparisons, $O(1)$ swaps
- On average: $O(n^2)$ comparisons, $O(n^2)$ swaps

Stability in sorting algorithms

- A sorting algorithm is said to be stable if two objects with equal or same keys appear in the same order in sorted output as they appear in the input array to be sorted.
- Bubble sort and insertion sort are stable
- Selection sort is not stable