## Sorting

Sorting problem: Given a list $a$ with $n$ elements possessing a total order, return a list with the same elements in non-decreasing order.

The sorting problem is perhaps the most fundamental problem in algorithms.

We can sort any kind of element that can be compared (`int`, `float`, `str`). In other words, we require a total order on the elements.

There are many direct applications of sorting (catalogs, reports, file listings, etc.)



## Sorting as a tool

There are also many indirect applications of sorting. For instance, algorithms can often be made faster by first sorting the data.

duplicates2.py

```
def has_duplicates_sorted(a):
  for i in range(len(a)-1):
    if a[i] == a[i+1]:
      return True
  return False

def has_duplicates(a):
  return has_duplicates_sorted(sorted(a))
```

Sorting + linear time!



## Selection sort

It's easy to find the minimum of $n$ numbers:

```
def find_min_index(a):
  mindex = 0
  for k in range(1, len(a)):
    if a[k] < a[mindex]:
      mindex = k
  return mindex
```

This gives immediately a sorting algorithm:

- A list with zero or one element is already sorted.
- Otherwise, find the minimum element, and recursively sort the remaining $n - 1$ elements.
- Concatenate the minimum and the sorted remaining elements.



## Selection sort

selection0.py

```
def selection_sort(a):
  if len(a) <= 1:
    return a
  k = find_min_index(a)
  b = selection_sort(a[:k] + a[k+1:])
  return [a[k]]+b
```

What is the running time?

This implementation works, but it creates a lot of lists, copies a lot of data, and could cause a runtime stack overflow...

## In-place Sorting

Sorting problem: Given a list $a$ with $n$ elements possessing a total order, return a list with the same elements in non-decreasing order.

Often we no longer need the original, unsorted data.

In-place Sorting: Given a list $a$ with $n$ elements possessing a total order, rearrange the elements inside the list into non-decreasing order.

Saves a lot of memory for huge data. Ideally we want to do this without creating any other list.

In Python:

- `sorted(a)` returns a sorted copy of `a`.
- `a.sort()` sorts the list `a` in-place.



## In-place selection sort with iteration

Find index of minimum in `a[i:]`

selection2.py

```
def find_min_index(a, i):
  mindex = i
  for k in range(i+1, len(a)):
    if a[k] < a[mindex]:
      mindex = k
  return mindex

def selection_sort(a):
  n = len(a)
  for i in range(0, n-1):
    k = find_min_index(a, i)
    t = a[i]
    a[i] = a[k]
    a[k] = t
```

This uses only one list (in-place) and cannot have stack overflow, but the running time is still $O(n^2)$.



## In-place Selection sort

Find index of minimum in `a[i:]`

selection1.py

```
def find_min_index(a, i):
  mindex = i
  for k in range(i+1, len(a)):
    if a[k] < a[mindex]:
      mindex = k
  return mindex
```

Sort `a[i:]`

```
def selection_sort(a, i):
  if j - i <= 1:
    return
  k = find_min_index(a, i)
  t = a[i]
  a[i] = a[k]
  a[k] = t
  selection_sort(a, i+1)
```

← Tail recursion!



## Insertion sort

Let's do it the other way round: Sort $n-1$ elements first, then insert the last element into the sorted sequence.

insertion0.py

```
def insertion_sort(a):
  if len(a) <= 1:
    return a
  b = insertion_sort(a[:-1])
  k = sorted_linear_search(b, a[-1])
  b.insert(k, a[-1])
  return b
```

## In-place insertion sort

insertion1.py

```
# sort a[:j]
def insertion_sort(a, j):
  if j <= 1:
    return
  insertion_sort(a, j-1)
  k = j-1      # remaining element index
  x = a[k]     # value of remaining element
  while k > 0 and a[k-1] > x:
    a[k] = a[k-1]
    k -= 1
  a[k] = x
```

This is not tail-recursion, but we can still easily make it iterative.



## Iterative in-place insertion sort

insertion2.py

```
def insertion_sort(a):
  for j in range(2, len(a)+1):
    # a[:j-1] is already sorted
    k = j-1        # remaining element index
    x = a[k]       # value of remaining element
    while k > 0 and a[k-1] > x:
      a[k] = a[k-1]
      k -= 1
    a[k] = x
```

Loop invariant allows us to argue the correctness of the program.



## Bubble Sort

Similar to selection sort, we bring the largest element to the end:

bubble1.py

```
def bubble_sort(a):
  for last in range(len(a), 1, -1):
    # bubble max in a[:last] to a[last-1]
    for j in range(last-1):
      if a[j] > a[j+1]:
        t = a[j]
        a[j] = a[j+1]
        a[j+1] = t
```

Bubble-up phase

If nothing happens during a bubble-up phase, we are done!



## Bubble sort with early termination

We stop when nothing happens in one phase.

```
def bubble_sort(a):
  for last in range(len(a), 1, -1):
    # bubble max in a[:last] to a[last-1]
    flipped = False
    for j in range(last-1):
      if a[j] > a[j+1]:
        flipped = True
        t = a[j]
        a[j] = a[j+1]
        a[j+1] = t
    if not flipped:
      return
```

Effective if the list is already (nearly) sorted.

What is the worst case running time?