Another Example with Lists: Selection Sort
Sorting

- *Sorting* and *searching* are the two most commonly performed operations by computer programs.
- You might have seen sorting in the context of spreadsheets, where we want to sort by a certain column.
- Sorting occurs commonly in more complicated contexts as well – graphics programs might maintain collections of polygons in 3-dimensional space in “sorted” order so as to render scenes efficiently.
Since sorting is such a common operation, there are many known sorting algorithms. (Quick sort, Merge sort, Heap sort, Selection sort, Insertion sort, Bubble sort, Shell sort, ...)

Today we will study the selection sort algorithm.

This will serve three purposes:
- Provide an introduction to a fundamental computational task
- Provide more clues to Homework 3.
- Show that lists are different from all other data types we have seen thus far due to a property called mutability. We will focus on this issue in the next lecture.

It is worth pointing out that selection sort is terribly inefficient and you should not use it in general. We'll also study some of the more efficient sorting algorithms – merge sort and quick sort, later.
The Selection Sort Algorithm

- \( L \) is the list we want to sort. Let \( n = \text{len}(L) \).

- **In iteration 1,**
  - we find a smallest element in \( L[0..n-1] \) (i.e., the entire list) and “swap” it with the first element (\( L[0] \)) in \( L \).
  - Thus after iteration 1, \( L[0] \) has its final value. We can now work on \( L[1..n-1] \).

- **In iteration 2,**
  - we find a smallest element in \( L[1..n-1] \) and “swap” it with the second element (\( L[1] \)) in \( L \).
  - Thus after iteration 2, \( L[0..1] \) has its final values.
• Thus after i iterations, the prefix of the list $L[0..i-1]$ has its final value.

• In iteration $i+1$,
  ○ we find a smallest element in $L[i..n-1]$ and “swap” it with $L[i]$.
  ○ Thus after iteration $i+1$, $L[0..i]$ has its final value.

• We will be done after $n-1$ iterations.
def selectionSort(L):
    n = len(L)
    index = 0

    while index < n-1:
        # Finds the index of a smallest element in the range L[index..n-1]
        m = minIndex(L, index)
        m

        # Bring this smallest element to the "front" by swapping L[m] and L[index]
        swap(L, index, m)

    index = index + 1
The function `minIndex`

# Finds and returns the index of a smallest element in the range L[lowerBound..len(L)-1]
def minIndex(L, lowerBound):
    # Initializations: we assume that the first element in L[lowerBound..len(L)-1]
    # is smallest.
    minElement = L[lowerBound]
    indexOfMin = lowerBound

    # We then process the rest of the range starting from L[lowerBound+1]
    index = lowerBound + 1
    while index < len(L):
        if L[index] < minElement:
            minElement = L[index]
            indexOfMin = index

        index = index + 1

    return indexOfMin
The function **swap**

# Exchanges the elements indexed i and j in list L

def swap(L, i, j):
    temp = L[i]
    L[i] = L[j]
    L[j] = temp
A few remarks about the code

- Note that the function `swap` does not return anything.
- It communicates with `selectionSort` by modifying the list \( L \) in-place and having this effect be felt "outside."
- This type of communication between functions is possible because lists have a property called *mutability*.
- This property, its implications, and how it is implemented behind the scenes is the topic of the next lecture.
Timing Selection Sort

- It is easy to time `selectionSort` using the `time` module.

- Checkout `timeSelectionSort.py` on the course page.

- We generated random length-$n$ lists for $n = 1000, 2000, ..., 10000$.

- For each $n$, we generated 100 such lists and averaged the running time of selection sort over 100 runs.
Timing Selection Sort

- X-axis shows length of the list, in units of 1000.
- Y-axis shows average time (over 100 replicates) in seconds.