Sorting with an Ordered Dictionary

We can use an ordered dictionary to sort a set
of values:

1. Add the elements one by one to the ordered dictionary with a series of put operations
2. Remove the elements in sorted order with a series of smallest() and remove() operations

The running time of this sorting method depends
on the ordered dictionary implementation

Sorting with an Ordered Dictionary

```
Algorithm OrdDict-Sort(A,n)
Input: Array A storing n values
Out: Array A sorted increasingly
    P ← empty ordered dictionary
    for i ← 0 to n-1 do
        P.put(A[i])
    for i ← 0 to n-1 do {
        e ← P.smallest()
        P.remove(e)
        A[i] ← e
    }
    return A
```

Selection Sort

If in *OrdDict-Sort* we implement the ordered dictionary using an unsorted array, the resulting algorithm is called *selection sort*.

Time complexity:

Each put operation takes O(1) time since we can insert the item at the end of the array.

Each smallest and remove operations take O(n) time since we have to traverse the entire array to find the smallest key and remove it.

Since the algorithm performs n put operations, n smallest operations, and n remove operations, the time complexity is $O(n^2)$

Ord-Dict Sort

If in *OrdDict-Sort* we implement the ordered dictionary using an AVL tree or a (2,4) tree then the complexity is reduced to O(n log n).

Time complexity:

Each put operation takes O(log n).

Each smallest and remove operations take O(log n).

Since the algorithm performs n put, n smallest, and n remove operations the time complexity is O(n log n)

Insertion Sort

If *OrdDict-Sort* we implement the ordered dictionary using a sorted array, the resulting algorithm is called *insertion sort*.

Time complexity:

Each put operation takes O(n) time since we need to insert the item in its correct position in the sorted array.

Each smallest and remove operations take O(1) time since the smallest value is at the beginning of the array. Note that we can store the index of the first value of the array and the index of the last one. Then, to remove the smallest value in the array we only need to increase the index of the first value.

Since the algorithm performs n put, n smallest, and n remove operations the time complexity is $O(n^2)$

In-Place Insertion Sort

We can implement selection-sort and insertion-sort without an external data structure; those versions of the sorting algorithms are said to be in-place. Here is in-place insertion-sort:

```
Algorithm insertion-sort (A,n)
In: Array A storing n values
Out: Array A sorted in increasing value
for i ← 1 to n-1 do {
    t ← A[i]
    j ← i -1    // Insert A[i] in the correct position inside the sorted array A[0, ..., i-1]
    while (j >= 0) and (A[j] > t) do {
        A[j+1] ← A[j]
        j ← j-1
    }
    A[j+1] ← t
}
return A
```

In-Place Selection Sort

We can implement selection-sort and insertion-sort without an external data structure; those versions of the sorting algorithms are said to be in-place. Here is in-place insertion-sort:

```
Algorithm insertion-sort (A,n)
In: Array A storing n values
Out: Array A sorted in increasing value
for i ← 0 to n-2 do {
    smallest ← i   // Find the smallest value in A[i, ..., n-1]
    for j ← i+1 to n-1 do
        if A[j] < A[smallest] then smallest ← j
    t ← A[smallest]
    A[smallest] ← A[i]
    A[i] ← t
}
return A
```